12 Advanced Topics

Many of the subjects covered in this chapter may provide benefits for all levels of programmers.

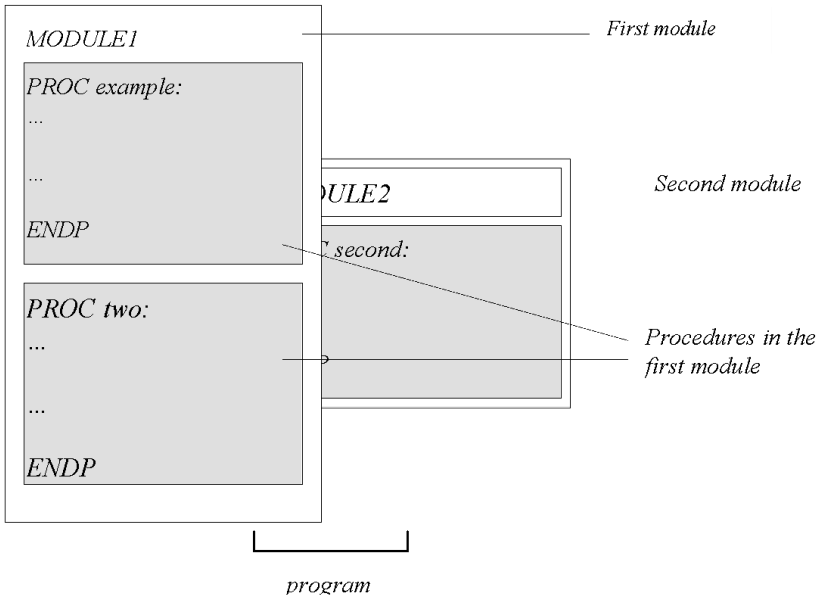
However, the subjects become progressively more technical. This manual cannot cover every OPL keyword in detail, as some give access to the innermost workings of the Psion. Some of these keywords are touched on in this chapter.

PROGRAMS, MODULES, PROCEDURES

Programs using more than one module

Program is the more general word for the “finished product” - a set of procedures which can be run. A program may consist of one module containing one or more procedures:

It may also consist of a number of modules containing various procedures:



A procedure in one module can call procedures in another module, provided this module has been loaded with the LOADM command. LOADM needs the filename of the module to load.

For example, if an OPL module `MAIN` has these two procedures:

```
PROC main:
  LOADM “library”
  pthis:                    REM run pthis: in this module
  pother:                   REM run pother: in “library”
  PRINT “Finished”
  PAUSE 40
  UNLOADM “library”
ENDP
```

```
PROC pthis:
  PRINT "Running pthis:"
  PRINT "in this module"
  PAUSE 40
ENDP
```

and a module called `LIBRARY` has this one:

```
PROC pother:
  PRINT "Running pother:"
  PRINT "in module called LIBRARY"
  PAUSE 40
ENDP
```

then running `MAIN` would display `Running pthis:` and `in this module`; then display `Running pother:` and `in module called LIBRARY`; and then display `Finished`.

You would usually only need to load other modules when developing large OPL programs.

Î You can use LOADM up to seven times, to load up to seven modules.

Ì You can use LOADM up to three times, to load up to three modules.

If you want to load any further modules, you must first unload one of those currently loaded, by using UNLOADM with the filename. To keep your program tidy, you should UNLOADM a module as soon as you have finished using it.

If there is an error in the running program, you will only be positioned in the Program editor at the place where the error occurred if you were editing the module concerned (and you ran the program from there).

In spite of its name, LOADM does not "load" the whole of another module into memory. Instead, it just loads a block of data stored in the module which lists the names of the procedures which it contains. If such a procedure is then called, the procedure will be searched for, copied from file into memory and the procedure will then be run.

The modules are searched through in the order that they were loaded, with the module loaded last searched through last. **You may make considerable improvements in speed if you as few modules as possible loaded at a time (so avoiding a long search for each procedure) and if you load the modules with the most commonly called procedures first.**

Cacheing procedures for speed

Î **On the Series 5, procedures are automatically cached.**

Ì On the Series 3c, without procedure cacheing, procedures are loaded from file whenever they are called, and discarded when they return. This is true even when procedures are all in one module. (With more than one module, LOADM simply loads a map listing procedure names and their positions in the module file so that they can be loaded fairly efficiently. It does not load procedures into memory.)

If a well-designed program calls procedures regularly, you can speed it up by cacheing procedures. This keeps the code for a procedure loaded in memory after it returns, so that if it is called again there is no need to fetch it from file again. The CACHE command takes two arguments the initial size of the cache and the maximum size (both in bytes). These can be up to 32,767 bytes. The minimum in both cases is 2000 bytes.

For a small program, you might use `CACHE 2000,2000` at the start of the program. Up to 2000 bytes of procedure code will be cached. If the cache fills up, and a procedure is called which is not in the cache, space will be made for it in the cache by removing other procedures from it.

For a much larger program, you might use `CACHE 10000,10000`. You may wish to change the settings and find the smallest setting which produces the maximum speed improvement.

Once a cache has been created, `CACHE OFF` prevents further cacheing, although the cache is still searched when calling subsequent procedures. `CACHE ON` may then be used to re-enable cacheing.

Calling procedures by strings

Procedures can be called using a string expression for the procedure name. Use an `@` symbol, optionally followed by a character to show what type of value is returned for example, `%` for an integer. Follow this with the string expression in brackets. You can use upper or lower case characters.

Here are examples showing the four types of value which can be returned:

`i% = @%(name$):(parameters)`	for integer
`l& = @&(name$):(parameters)`	for long integer
`s$ = @$(name$):(parameters)`	for string
`f = @(name$):(parameters)`	for floating-point

So, for example, `test$:(1.0,2)` and `@$("test"):(1.0,2)` are equivalent.

Note that the string expression does not itself include a symbol for the type (`%`, `&` or `$`).

You may find this useful if, in a more complex program, you want to "work out" the name of a procedure to call at a particular point. The chapter on 'Friendlier interaction' includes an example program which uses this method.

Where files are kept

On the Series 3c the Siena and the Series 5, the internal memory and disks (memory disks on the Series 5; SSDs on the Series 3c) use a DOS-compatible directory structure, the same as that used on disks on PCs. However, the two machines differ quite considerably in this area as described below.

Where files are kept on the Series 5

To specify a file completely, the Series 5 uses a *drive* (or *device*), *folder* and *filename*:

- The drive is the area on the Series 5 where the file is kept. This can be `C:` (internal disk), `D:` (memory disk drive) `E:`,..., `Y:` or `Z:` (ROM).
- Every drive has one *root folder*, usually written as a backslash (`\`). This can "contain" files and/or other folders, each of which can contain more files and/or more folders. When you have "folders in folders" like this, they're often called *subfolders*. Their names show where they are. For example, the root folder (`\`) could contain a folder called `\JO`, which might in turn contain a folder called `\JO\BACKUP`, which might contain some files.

- Filenames are composed of up to 256 characters (but see below), optionally followed by a *file extension* comprised of a dot and from one to three characters. A filename may not begin with a back or forward slash (\ or /) or a colon (:). Any trailing zeros on the filename are stripped.

  The Series 5 filing system does not treat extensions as a special component of a filename except when parsing (i.e. for PARSE$) where the extension is treated as on the Series 3c (see below). So while `myfile.` on the Series 3c meant the same as `myfile`, on the Series 5 the dot is actually part of the name. You can also use long filenames with embedded spaces and any number of dots like: `OPL Reference manual` or `Very.long.filename`.

To specify a file completely, you add the three parts together. The folder part must end with a backslash. So an OPL module named `TEST`, in a folder called `\JO` in the Series 5 internal memory can

be specified as C:\JO\TEST. If this file were in the \JO\BACKUP folder, it would be completely specified as C:\JO\BACKUP\TEST. If it were in the root folder, you would specify it as C:\TEST.

A full file specification may be up to 255 characters long for OPL.

In OPL, as in other applications, the files are kept on the drive and in the folder you specify.

Using file specifications in OPL

OPL commands which specify a filename, such as OPEN, CREATE, gLOADBIT and so on, can also use any or all of the other parts that make up a full file specification. (Normally, the only other part you might use is the drive name, if the file were on a memory disk.) So for example, OPEN "C:\ADDR.TXT" tries to open a file called ADDR.TXT in the root folder of the internal disk.

You can use the PARSE$ function if you need to build up complex filenames. See the 'Alphabetic Listing' chapter for more details of PARSE$.

Current folder

The current folder for all commands is always C:\ unless it has been changed by the command SETPATH. Hence any use of a keyword which takes a filename as an argument will only look in the current folder and so if this is other than C:\, it should be specified either by SETPATH or by including it in the filename. For example, to check whether the file Program1 in the directory D:\MyPrograms\ exists, either

```
SETPATH "d:\MyPrograms\"
...
IF EXISTS ("Program1")
...
```

or

```
IF EXISTS ("d:\MyPrograms\Program1")
...
```

Control of folders

Use the MKDIR command to make a new folder. For example, MKDIR "C:\MINE\TEMP" creates a C:\MINE\TEMP folder, also creating C:\MINE if it is not already there. An error is raised if the chosen folder exists already. Use TRAP MKDIR to avoid this.

SETPATH sets the current folder for file access - for example, SETPATH "C:\DOCUMENTS". LOADM continues to use the folder of the running program, but all other file access will be to the newly specified folder.

Use RMDIR to remove a folder - for example, RMDIR "C:\MINE" removes the MINE folder on C:. A 'Does not exist' error is raised if the folder does not exist. Use TRAP RMDIR to avoid this. A '*File* is in use' error will result if you try to remove a folder which contains open files.

Where files are kept on the Series 3c and Siena

Note that any mention of SSDs in this section refers to the Series 3c only. The Siena does not have an SSD drive. Other discussion refers to both machines as usual.

The Series 3c hides the complexities of the directory structure. You have only to supply a filename, and to say whereabouts a file is stored - internally, on an SSD or on another computer to which the Series 3c is linked.

In fact, in order to specify a file completely, the Series 3c uses a *filing system*, *drive* (or *device*), *directory* and *filename*:

- The filing system usually specifies the computer, and is usually LOC:: ('local' the Series 3c) or REM:: ('remote' an attached computer). This is always three letters and two colons.

- The drive is the area on that computer where the file is kept. On the Series 3c this can be `M:` (internal disk), `A:` (left SSD drive) or `B:` (right SSD drive).
- Every drive has one *root directory*, usually written as a backslash (`\`). This can "contain" files and/or other directories, each of which can contain more files and/or more directories. When you have "directories in directories" like this, they're often called *subdirectories*. Their names show where they are. For example, the root directory (`\`) could contain a directory called `\JO`, which might in turn contain a directory called `\JO\BACKUP`, which might contain some files.

- Filenames are composed of one to eight letters and/or numbers, optionally followed by a *file extension* comprised of a dot and from one to three letters/numbers. File extensions are by convention used to group different types of files. The Series 3c uses file extensions in this way, but hides this from you.

To specify a file completely, you add the four parts together. The directory part must end with a backslash. So an OPL module named `TEST`, in a directory called `\JO` in the Series 3c internal memory can be specified as `LOC::M:\JO\TEST.OPL`. If this file were in the `\JO\BACKUP` directory, it would be completely specified as `LOC::M:\JO\BACKUP\TEST.OPL`. If it were in the root directory, you would specify it as `LOC::M:\TEST.OPL`.

A full file specification may be up to 128 characters long.

In OPL, unless you say otherwise, files are kept on the Series 3c (`LOC::`), in the internal memory (`M:`). The directories and file extensions used are:

Type of file	*Directory*	*File extension*
OPL modules	\OPL	.OPL
translated modules	\OPO	.OPO
data files	\OPD	.ODB
bitmaps	\OPD	.PIC

Using file specifications in OPL

OPL commands which specify a filename, such as OPEN, CREATE, gLOADBIT and so on, can also use any or all of the other parts that make up a full file specification. (Normally, the only other part you might use is the drive name, if the file were on an SSD.) So for example, `OPEN "REM::C:\ADDR.TXT"...` tries to open a data file called `ADDR.TXT` on the root directory of a hard disk `C:` on an attached PC.

You can use the PARSE$ function if you need to build up complex filenames. See the 'Alphabetic Listing' chapter for more details of PARSE$.

Unless you have a good reason, though, it's best not to change directories or file extensions for files on the Series 3c. You can lose information this way, unless you're careful.

Control of directories

Use the MKDIR command to make a new directory. For example, `MKDIR "M:\MINE\TEMP"` creates a `M:\MINE\TEMP` directory, also creating `M:\MINE` if it is not already there. An error is raised if the chosen directory exists already - use `TRAP MKDIR` to avoid this.

SETPATH sets the current directory for file access - for example, `SETPATH "A:\DOCS"`. LOADM continues to use the directory of the running program, but all other file access will be to the new directory instead of `\OPD`.

Use RMDIR to remove a directory - for example, `RMDIR "M:\MINE"` removes the `MINE` directory on `M:\`. An error is raised if the directory does not exist use `TRAP RMDIR` to avoid this.

You can only remove empty directories.

File specifications on REM::

You should not assume that remote file systems use DOS-like file specifications for example, an Apple Macintosh specification is `disk:folder:folder:filename`. You can only assume that there will be four parts - disk/device, path, filename and (possibly null) extension. PARSE$, however, will always build up or break down REM:: file specifications correctly.

SAFE POINTER ARITHMETIC

Î Note that on the Series 5, if you are using 32-bit addressing, as will be the case by default (see the '32-bit addressing' section later in this chapter), you should use ordinary long integer arithmetic and should **not** use UADD and USUB.

However, **on the Series 3c and Siena or if you have used SETFLAGS on the Series 5 to enforce the 64K memory limit**, whenever you wish to add or subtract something from an integer which is an address of something (or a pointer to something) you should use UADD and USUB. If you do not, you will risk an 'Integer overflow' error.

An address can be any value from 0 to 65535. An integer variable can hold the full range of addresses, but treats those values from 32768 to 65535 as if they were -32768 to -1. If your addition or subtraction would convert an address in the 0-32767 range to one in the 32768-65535 range, or vice versa, this would cause an 'Integer overflow'.

UADD and USUB treat integers as if they were unsigned, i.e. as if they really held a value from 0 to 65535.

For example, to add 1 to the address of a text string (in order to skip over its leading byte count and point to the first character in the string), use `i%=UADD(ADDR(a$),1)`, not `i%=ADDR(a$)+1`.

USUB works in a similar way, subtracting the second integer value from the first integer, in unsigned fashion for example, `USUB(ADDR(c%),3)`.

`USUB(x%,y%)` has the same effect as `UADD(x%,-y%)`.

OPL APPLICATIONS (OPAS)

You can make an OPL program appear as an icon in the system screen, and behave like the other icons there, by converting it into an *OPL application*, or *OPA*. The support for OPAs on the Series 5 is changed and extended from that of the Series 3c and Siena. The following two sections deal separately with these systems.

OPAS ON THE SERIES 5

There are two settings for OPAs which are set using the FLAGS command (similar to TYPE on the Series 3c):

- `FLAGS 1` is used if your application can create files. It will then be included in the list of applications offered when the user creates a new file from the System screen (like Program, Word etc.).
- `FLAGS 2` prevents the application from appearing in the Extras bar in the System screen. It is very unusual to have this flag set.

Once created, OPAs may be used in the same way as the built in Series 5 applications: new document files may be created from the System screen using 'New File' if the flag 1 is set and/or from the Extras bar if the flag 2 is **not** set. Existing files may be opened by selecting them on the System screen as usual. You can stop a running OPA by using the Task list.

Defining an OPA

To make an OPA, your OPL file should **begin with** the APP keyword, followed by a name for the OPA in the machine's default language and its UID. The name may be up to 250 characters. (Note that it does not have quote marks.) If the CAPTION command is used, however, this default name will be discarded (see below).

The UID specifies the UID of the application. For applications that are to be distributed, UIDs are reserved by contacting Psion. These official UIDs are guaranteed to be unique to the application and have values of &10000000 and above. To obtain a reserved UID you should contact Psion Software in one of the following ways:

e-mail to **uid_sw@software.psion.com**

or use the EPOC World web site, **http://software.psion.com/EPOCWorld/**

or fax to **+44-171-724-4048, attn UID Allocations**

or write to **UID Allocations, Psion Software plc, 19 Harcourt St, London W1N 1DT, England**

For applications developed for personal use there is no need to reserve official UIDs, and any UID between &01000000 and &0fffffff may be selected. However, if the UIDs of two different applications did clash, then either your application's documents will have the wrong icon and selecting these will run the other application, or the other application's documents will seem to belong to your application.

The APP line may be followed by any or all of the keywords CAPTION, ICON and FLAGS. Finally, use ENDA, and then the first procedure may begin as in a normal OPL file. Here is an example of how an OPA might start:

```
APP Expenses,&20000000        REM !!! Should be a reserved UID !!!
  FLAGS 1
  ICON "icon.mbm"
  CAPTION "Expenses",1        REM English name
  CAPTION "Expenses",10       REM American name
ENDA
```

Here is another example:

```
APP Picture,&30000000         REM !!! Should be a reserved UID !!!
  FLAGS 2
  ICON "picicon.mbm"
ENDA
```

FLAGS takes an integer argument of 1 or 2. The meanings of these are outlined above. If you don't specify the flags, none are set.

ICON gives the name of a *multi-bitmap file* (MBM), also known as an *EPOC Picture file*, which contains the icons for an OPL application. These icons are used on the Extras bar and for the application's documents on the System screen. The multi-bitmap file can contain up to three bitmaps of sizes 24×24, 32×32 and 48×48 pixels respectively, **but each must be paired with a mask**. Each bitmap should be followed by its mask in the multi-bitmap file, so that bitmaps and masks occur alternately. The pixels which are set in the mask specify pixels in the bitmap which are to be used. Pixels which are clear in the mask specify pixels that are not to be used from the bitmap, allowing the background to be displayed in these pixels. Only pixels which are set in the mask are drawn in the final icon.

The different sizes of bitmaps are used for the different zoom levels in the System screen. The sizes are read from the MBM and the most suitable size is zoomed if the exact sizes required are not provided or if some are missing.

You can use ICON more than once within the APP...ENDA construct. The translator only insists that all icons are paired with a mask of the same size in the final ICON list. This allows you to use MBMs containing just one bitmap, as produced by the Sketch application. Icons may also be created on a PC (if the appropriate tools are available), or of course by another OPL program or application.

CAPTION specifies an application's *public name* (or *caption*) for a particular language, which is the name which will appear below its icon on the Extras bar and in the list of 'Programs' in the 'New File' dialog (assuming the setting of FLAGS allows these) when that is the language used by the machine. This name is also used as the default document name for documents launched using the Extras bar. The maximum length of the caption is 255 characters. However, note that a caption no longer than around 8 letters will look best on the Extras bar. If any CAPTION statement is included in the APP…ENDA structure, then the default caption provided by the APP declaration will be discarded. Therefore as many statements of CAPTION as are necessary to cover all the languages required, including the language of the machine on which the application is originally developed, should be used. The constants for the language codes are supplied in Const.oph. See the 'Calling Procedures' chapter for details of how to use this file and Appendix E for a listing of it.

On creation of an application, a folder specifically for the application is also created. For example, if the OPA has the name `AppXxx` the folder created will be `\System\Apps\AppXxx\`. The APP file itself (the translated OPL module) and the *application information file* (AIF) which contains three icon and mask pairs and the application caption and flags, are created in this folder. For example, the AIF for `AppXxx.app` would be `\System\Apps\AppXxx\AppXxx.aif`. The AIF file will be generated based on all the information contained in the APP…ENDA construct, but if any information is missing defaults will be used. These are:

- for ICON: the default (question mark) icons
- for CAPTION: the caption specified in the APP declaration
- for FLAGS: the default value of 0.

The arguments to any of the keywords between APP and ENDA must be constants and not expressions.

Running the OPA

Once you've successfully translated the OPL file, the applications icon will automatically appear on the Extras bar and/or on the 'Program' list in the 'New File' dialog (as specified by the FLAGS setting). The new documents of the application may then be created and existing ones opened as with other built in Series 5 applications.

The first thing a file-based OPA should do is to get the name of the file to use, and check whether it is meant to create it or open it. `CMD$(2)` returns the full name of the file to use; `CMD$(3)` returns "C" for "Create", "O" for "Open" or "R" for "Run".

All document-based OPAs should handle all of these cases; for example, if a "Create" fails because the file exists already, or an "Open" fails because it does not, OPL raises the error, and the OPA should take suitable action. Note however, that in general the System screen will not allow such events to occur and therefore they are unlikely to happen.

"R" means that your application has been run from the OPL Program editor or has been selected via the application's icon on the Extras bar, and not by the selection or creation of one of its documents from the system screen. A default filename, including path, is passed in `CMD$(2)`. When "R" is passed, an application should always try to open the last-used document. This is the document that was in use when the application was last closed, **not** the document that was most recently opened. The name of this document should be stored in a `.INI` file with the same name and in the same folder as your application. So for example, `AppXxx.app` would have `.INI` file `\System\Apps\AppXxx\AppXxx.ini`. If the `.INI` file does not exist or cannot be opened for any reason, or if the document listed there no longer exists, you should create the document named in `CMD$(2)`. `CMD$(2)` is a default name provided by the System based on your application's caption. If the `.INI` file is corrupt it should be deleted before going on to create `CMD$(2)`. No error message should be displayed in this case.

Constants for the array indices of CMD$ and the return values of `CMD$(3)` are supplied in Const.oph. See the 'Calling Procedures' chapter for details of how to use this file and Appendix E for a listing of it.

How the Series 5 talks to an OPA

When the Series 5 wants an OPA to exit or to switch files, it sends it a *System message*, in the form of an event. For example, this would happen if the 'Close file' option in the Task list were used to stop a running OPA.

TESTEVENT and GETEVENT32 (synchronous) and GETEVENTA32 (asynchronous) check for certain events, including both keypresses and System messages. All types of OPA **must** use these keywords to check for **both** keypresses and System messages; keyboard commands such as GET, KEY and KEYA cause other events to be **discarded**.

GETEVENT32 waits for an event whereas TESTEVENT simply checks whether an event has occurred without getting it. If TESTEVENT returns non-zero, an event has occurred, and can be read with GETEVENT32. However, it is recommended that you use either GETEVENT32 or GETEVENTA32 without TESTEVENT as **TESTEVENT may use a lot of power, especially when used in a loop** as will often be the case. GETEVENT32 may be used if you only wish to pick up window server events, but otherwise you should use the asynchronous GETEVENTA32. TESTEVENT should generally only be used when polling while doing something else in background.

GETEVENT32 and GETEVENTA32 both take one argument: the name of a long integer array, for example, `GETEVENT32 ev&()`. The array should contain at least 16 long integers.

For example, if the event is a keypress `(ev&(1) AND &400) = 0` and,

`ev&(1)` = keycode (as for GET)

`ev&(2)` = time stamp (gives the time of the keypress)

`ev&(3)` = scan code (locates the key on the keyboard)

`ev&(4)` = modifier code (e.g. Shift, Control)

`ev&(5)` = repeat. Note that this is strictly the repeat value, i.e. if there is only one keypress, then the value of `ev&(5)` is 0.

For non-key events `(ev&(1) AND &400)` will be non-zero. On the Series 5, these include *pointer events* (pen events). For an application, it may be suitable to filter out certain pointer events. This may be done using the POINTERFILTER command. POINTERFILTER takes two arguments: a filter and a mask. Each of the bits in these two arguments represents a certain pointer event, allowing a flag to be set to say whether that event should be filtered out. The bits which are set in the mask specify bits in the filter which are to be used. Bits which are clear in the mask specify bits that are not to be used from the filter. This makes it possible to filter out some pointer events, filter back in some others and leave the settings of some alone all in one call to POINTERFILTER. To set or clear flags you would set and clear them in the filter and set all the flags that require changing in the mask. To leave some bits in the filter alone just don't set the bits in the mask. See the 'Alphabetic Listing' chapter for more details of POINTERFILTER and the values returned to `ev&()` by GETEVENT32 and GETEVENTA32.

If the event is a System message to change files or quit, `ev&(1)=&404`. You should then use GETCMD$ to find the action required.

GETCMD$ returns a string, whose first character is "C", "O" or "X".

You can only call GETCMD$ once for each event. You should do so as soon as possible after reading the event. Assign the value returned by GETCMD$ to a string variable so that you can extract its components.

If you have `c$=GETCMD$`, the first character, which you can extract with `LEFT$(c$,1)`, has the following meaning:

"C" - close the current document, and create the specified new file.

"O" - close the current document, and open the specified existing file.

"X" - save the name of the current document (if any) to the `.INI` file, close the current document and quit the OPA.

Again with `c$=GETCMD$, MID$(c$,2,255)` is the easiest way to extract the filename.

Note that events are ignored while you are using keywords which pause the execution of the program GET, GET$, EDIT, INPUT, PAUSE, MENU and DIALOG. If you need to use these keywords, use `LOCK ON` / `LOCK OFF` (described later) around them to prevent the System screen from sending messages.

System screen compliance

A well-behaved Series 5 application should obey the following important guidelines:

- All applications should respond in the standard way to the initial command-line and to any System screen messages to switch files. See the 'How the Series 5 talks to an OPA' section above.
- All applications should support a toolbar. Toolbar.opo is supplied in the ROM and provides the set of procedures required for this support. See the 'Friendlier Interaction' chapter.
- All Series 5 applications should save non-document files (external files) to their so-called *application directory*. For an application called `AppXxx`, the application directory is `\System\Apps\AppXxx\`. This is also the directory where the application itself is saved.

  You can find out the full file specification of external files by using PARSE$. For example,

  ```
  p$=PARSE$("NEW",LEFT$(CMD$(1),LEN(CMD$(1)-4),x%())
  ```

  `CMD$(1)` gives the device and the path of the OPL application and the name of the file.

  By default the system screen hides the System folder and its subfolders. Use the 'Preferences' option in the 'Tools' menu (Ctrl+K) in the System screen and check the checkboxes for both 'Show hidden files' and 'Show 'System' folder' to show all applications and their files.

  - The FLAGS keyword in the APP...ENDA construct should have the value 1 if the application supports *documents* (a file that will run your application when selected from the System screen). Without this flag the application will not be listed when the user chooses to create a new file from the System screen (see sections above).
- You should set ESCAPE OFF so that it is impossible for the user to stop the application running by hitting Ctrl+Esc.

Launching help files

It is possible to write your own Help for your own application, which should be created in the Data application. The file should be similar to the built-in help, using two labels only for each entry to give a title and the explanation itself. The file should be stored in the application's folder with the filename extension `.hlp`. The procedure RUNAPP&: in System OPX (see the 'Using OPXs on the Series 5' chapter for more details) may then be used to launch Data and open the application help file. For example, if you want to run a help file called `Myapp.hlp`, the help file for your own application called `Myapp`, you should use:

```
drv$=LEFT$(CMD$(1),2)
RUNAPPP&:("Data",drv$+"\System\Apps\Myapp\Myapp.hlp","",0)
```

You should make it possible for users of your application to launched your help file from the application's menus. Following the Series 5 style guide, the 'Help' item should appear on the 'Tools' menu at the bottom of the section above 'Infrared', so it will be followed by a separating line. The shortcut key should be Ctrl+Shift+H.

Example OPAs

Here is an OPA which just prints the keys you press. It is not a document-based application (like Calc, but unlike Word which is document-based) so it uses `FLAGS 0`. The keyboard procedure `getk&:` returns the key pressed, as with GET, but jumps to a procedure `endit:` if a System message to close down is received. (OPAs with flags set to 0 do not receive "change file" messages.)

```
CONST KUidAppMyApp0&=&40000000 REM !!! Should be a reserved UID!!!
APP myapp0,KUidAppMyApp0&
  CAPTION "Get Key",1
  ICON "myapp0.mbm"
ENDA

PROC start:
  GLOBAL a&(10),k&
  FONT 11,16
  PRINT "Q to Quit"
  PRINT " or select 'Close file'"
  PRINT " from the Task List"
  DO
    k&=getk&:
    PRINT CHR$(k&);
  UNTIL (k& AND &ffdf)=%Q          REM Quick way to do uppercase
ENDP

PROC getk&:
  DO
    GETEVENT32 a&()
    IF a&(1)=&404
      IF LEFT$(GETCMD$,1)="X"
        endit:
      ENDIF
    ENDIF
  UNTIL a&(1)<256
  RETURN a&(1)
ENDP

PROC endit:
  STOP
ENDP
```

Here is a similar document-based OPA. It does the same as the previous example, but System messages to change files cause the procedure `fset:` to be called. The relevant files are opened or created. A proper application with a toolbar would call `TBarSetTitle:` to change its title. See the 'Friendlier Interaction' chapter.

```
CONST KUidAppMyApp1&=&50000000 REM !!! Should be a reserved UID!!!
```

```
APP myapp1,KUidAppMyApp1&
  FLAGS 1
  CAPTION "Get Key Doc",1
  ICON "myapp1.mbm"
ENDA

PROC start:
  GLOBAL a&(10),k&,w$(255)
  FONT 11,16 :w$=CMD$(2)
  fset:(CMD$(3))
  PRINT "Q to Quit"
  PRINT "use the Task list"
  PRINT "to create/switch files"
  DO
    k&=getk&:
    PRINT CHR$(k&);
  UNTIL (k& AND &ffdf)=%Q
ENDP

PROC getk&:
  LOCAL t$(1)
  DO
    GETEVENT32 a&()
    IF a&(1)=$404
      w$=GETCMD$
      t$=LEFT$(w$,1)
      w$=MID$(w$,2,255)
      IF t$="X"
        endit:
      ELSEIF t$="C" OR t$="O"
        TRAP CLOSE
        IF ERR
          GIPRINT ERR$(ERR)
          CONTINUE
        ENDIF
        fset:(t$)
      ENDIF
    ENDIF
```

```
  UNTIL a&(1)<256
  RETURN a&(1)
ENDP

PROC fset:(t$)
  LOCAL p&(6)
  IF t$="C"
    TRAP DELETE w$
    SETDOC w$
    TRAP CREATE w$,A,A$
  ELSEIF t$="O"
    SETDOC w$
    TRAP OPEN w$,A,A$
  ENDIF
  IF ERR
    CLS :PRINT ERR$(ERR) REM should revert to old file if possible
    GET :STOP
  ENDIF
ENDP

PROC endit:
  STOP
ENDP
```

You should, as in both these examples, be precise in checking for the System message; if in future the GETCMD$ function were to use values other than “C”, “O” or “X”, these procedures would ignore them.

If you need to check the modifier keys for the returned keypress, use `a&(4)` instead of KMOD.

SETDOC called just before the creation of the file ensures that the created file is a *document*, i.e. that the file launches its associated application when selected. The strings passed to SETDOC and CREATE (or gSAVEBIT) should be exactly the same, otherwise a non-document file will be created. SETDOC should also be called when opening a document to allow the System screen to display the correct document name in its task list. OPL explicitly supports database and multi-bitmap files as documents.

To be strict, whenever **creating** a file, an OPA should first use PARSE$ to find the disk and directory requested. It should then use `TRAP MKDIR` to ensure that the directory exists.

When an OPA cannot respond

The LOCK command marks an OPA as locked or unlocked. When an OPA is locked with `LOCK ON`, the System will not send it events to change files or quit. If, for example, you attempt to close down the OPA from the Task list, a message will appear, indicating that the OPA cannot close down at that moment.

You should use `LOCK ON` if your OPA uses a keyword, such as MENU, DIALOG and EDIT, which pauses the execution of the program. You might also use it when the OPA is about to go busy for a

considerable length of time, or at any other point where a clean exit is not possible. Do not forget to use `LOCK OFF` as soon as possible afterwards.

An OPA is initially unlocked.

OPAs on the Series 3c and Siena

There are five different types of OPA, called type 0 to type 4:

- `TYPE 0` (like Calc): The OPA uses no files.
- `TYPE 1`: Only one file is used. A type 1 OPA will look the same as a type 0. The only difference is that the type 1 is using a file, of the same name as the OPA.
- `TYPE 2` (like World): You can have more than one file, but only one can be in use (bold) at any time.

  When you pick a new file to use, its name becomes bold, and the one that was previously bold reverts to normal. **What has actually happened is that the running OPA has switched files** - it has **not** closed down, and no new copy of the OPA is run.
- `TYPE 3` (like Data, Word, Agenda, Sheet): You can have more than one file, and any number may be open (bold) at a given time.

  When you select a new file, one of the running OPAs normally switches to this file, as with type 2 OPAs. You can, however, with Shift-Enter, start a new OPA running just for this file, without a different file exiting.
- `TYPE 4` (like RunOpl): Many files can be used, and any number may be in use at a given time. When you select a new file, a new version of the OPA is always run, to use the new file.

Types 3 and 4 allow more than one file to be **in use** (i.e. have their names in bold). When this happens **a separate version of the OPA runs for each bold file**. With types 0, 1 and 2, only one version of the OPA can be running at any time.

Initially, the OPA's name appears beneath the icon. If you move onto this name and press Enter, file-based OPAs (types 1 to 4) will use a file of this name. Types 2, 3 and 4 allow you to create lists of files below the icon (with the 'New file' option). You use the file lists in the same way as the lists under the other icons in the System screen.

You can stop a running OPA by moving the cursor onto its bold name and pressing Delete. After a 'Confirm' dialog, the System screen tells the OPA to stop running.

Defining an OPA

To make an OPA, your OPL file should **begin with** the APP keyword, followed by a name for the OPA. The name should begin with a letter, and comprise of 1 to 8 letters and/or numbers. (Note that it does not have quote marks.) The APP line may be followed by any or all of the keywords PATH, EXT, ICON and TYPE. **A Series 3c OPA should also add $1000 to the type if it has its own 48x48 pixel, black/grey icon** (see the discussion of ICON below for details). Finally, use ENDA, and then the first procedure may begin as in a normal OPL file. Here is an example of how an OPA might start:

```
APP Expenses
  TYPE $1003
  PATH "\EXP"
  EXT "EXP"
  ICON "\OPD\EXPENSES.PIC"
ENDA
```

Here is another example:

```
APP Picture
  TYPE 1
```

```
ENDA
```

TYPE takes an integer argument from 0 to 4. The various types of OPA are outlined earlier. If you don't specify the type, 0 is used.

PATH gives the directory to use for this OPA's files. If you do not use this, the normal `\OPD` directory will be used. The maximum length, including a final `\`, is 19 characters. Don't include any drive name in this path.

EXT gives the file extension of files used by this OPA. If you do not specify this, `.ODB` is used. Note that the files used by an OPA do not have to be data files, as the I/O commands give access to files of all kinds. EXT does not define the file type, just the file extension to use. However, **for simplicity's sake, examples in this section use data files**.

(PATH and EXT provide information for the System screen - they do not affect the program itself. The System screen displays under the OPA icon all files with the specified extension in the path you have requested.)

ICON gives the name of the bitmap file to use as the icon for this OPA. If no file extension is given, .PIC is used. If you do not use ICON, the OPA is shown on the System screen with a standard OPA icon.

As mentioned above, you should add $1000 to the argument to TYPE for a Series 3c icon. This specifies that the icon has size 48x48 pixels (instead of 24x24 as it was on the Series 3). If the first bitmap has size 24x24, it is ignored and the following two bitmaps must be the 48x48 black and grey icons respectively. If the first bitmap is 48x48, it is assumed to be the black icon and the grey icon must follow. **If $1000 is not set, a scaled up 24x24 icon will be used.** The translator does not check the size of the icons. If you want to design your own icon using an OPL program, see gSAVEBIT for details on saving both black and grey planes to a bitmap file.

The arguments to any of the keywords between APP and ENDA must be constants and not expressions. So, for example, you must use `TYPE $1003` instead of `TYPE $1000 OR 3`.

Running the OPA

Once you've translated the OPL file, return to the System screen and use Install on the App menu to install the OPA in the System screen. (You only need to do this once.) Once installed, file-based OPAs are shown with the list of available files, if any are found. Otherwise, the name used after the APP keyword appears below the icon.

Note that the translated OPA is saved in a `\APP` directory. If you previously translated the module without the APP...ENDA at the start, the old translated version will still be listed under the RunOpl icon, and should be deleted.

The first thing a file-based OPA should do is to get the name of the file to use, and check whether it is meant to create it or open it. `CMD$(2)` returns the full name of the file to use; `CMD$(3)` returns "C" for "Create" or "O" for "Open". **All** file-based OPAs (types 1 to 4) should handle both these cases; if a "Create" fails because the file exists already, or an "Open" fails because it does not, OPL raises the error, and the OPA should take suitable action - perhaps even just exiting.

How the Series 3c talks to an OPA

When the Series 3c wants an OPA to exit or to switch files, it sends it a *System message*, in the form of an event. This would happen if you press Delete to stop a running OPA, or select a new file for a type 2 or 3 OPA.

TESTEVENT and GETEVENT check for certain events, including both keypresses and System messages. All types of OPA **must** use these keywords to check for **both** keypresses and System messages; keyboard commands such as GET, KEY and KEYA cause other events to be **discarded**.

GETEVENT waits for an event whereas TESTEVENT simply checks whether an event has occurred without getting it.

If TESTEVENT returns non-zero, an event has occurred, and can be read with GETEVENT. This takes one argument, the name of an integer array for example, `GETEVENT a%()`. The array should be at least 6 integers long. (This is to allow for future upgrades you only need use the first two integers.)

If the event is a keypress:

`a%(1)` = keycode (as for GET)

`a%(2) AND $00ff` = modifier (as for KMOD)

`a%(2)/256` = auto-repeat count (ignored by GET; you can ignore it too)

For non-key events `(a%(1) AND $400)` will be non-zero. If the event is a System message to change files or quit, `a%(1)=$404`. You should then use GETCMD$ to find the action required.

GETCMD$ returns a string, whose first character is “C”, “O” or “X”. If it is “C” or “O”, the rest of the string is a filename.

You can only call GETCMD$ once for each event. You should do so as soon as possible after reading the event. Assign the value returned by GETCMD$ to a string variable so that you can extract its components.

If you have `c$=GETCMD$`, the first character, which you can extract with `LEFT$(c$,1)`, has the following meaning:

“C” - close down the current file, and create the specified new file.

“O” - close down the current file, and open the specified existing file.

“X” - close down the current file (if any) and quit the OPA.

Again with `c$=GETCMD$`, `MID$(c$,2,128)` is the easiest way to extract the filename.

Note that events are ignored while you are using keywords which pause the execution of the program GET, GET$, EDIT, INPUT, PAUSE, MENU and DIALOG. If you need to use these keywords, use `LOCK ON` / `LOCK OFF` (described later) around them to prevent the System screen from sending messages.

Example OPAs

Here is a type 0 OPA, which just prints the keys you press. The keyboard procedure `getk%:` returns the key pressed, as with GET, but jumps to a procedure `endit:` if a System message to close down is received. (Type 0 OPAs do not receive “change file” messages.)

`getk%:` does not return events with values 256 ($100) or above, as they are not simple keypresses. This includes the non-typing keys like Menu ($100-$1FF), hot-keys ($200-$3FF), and non-key events ($400 and above).

```
APP myapp0
  TYPE $1000
  ICON "\opd\me"
ENDA


PROC start:
  GLOBAL a%(6),k%
  STATUSWIN ON :FONT 11,16
  PRINT "Q to Quit"
  PRINT " or press Delete in"
  PRINT " the System screen"
  DO
```

```
    k%=getk%:
    PRINT CHR$(k%);
  UNTIL (k% AND $ffdf)=%Q          REM Quick way to do uppercase
ENDP

PROC getk%:
  DO
    GETEVENT a%()
    IF a%(1)=$404
      IF LEFT$(GETCMD$,1)="X"
        endit:
      ENDIF
    ENDIF
  UNTIL a%(1)<256
  RETURN a%(1)
ENDP

PROC endit:
  STOP
ENDP
```

Here is a similar type 3 OPA. It does the same as the previous example, but System messages to change files cause the procedure `fset:` to be called. The relevant files are opened or created; the name of the file in use is shown in the status window.

```
APP myapp3
  TYPE $1003
  ICON "\opd\me"
ENDA

PROC start:
  GLOBAL a%(6),k%,w$(128)
  STATUSWIN ON :FONT 11,16 :w$=CMD$(2)
  fset:(CMD$(3))
  PRINT "Q to Quit"
  PRINT " or press Delete in"
  PRINT "the System screen"
  PRINT " or create/swap files in"
  PRINT "the System screen"
  DO
    k%=getk%:
```

```
    PRINT CHR$(k%);
  UNTIL (k% AND $ffdf)=%Q
ENDP

PROC getk%:
  LOCAL t$(1)
  DO
    GETEVENT a%()
    IF a%(1)=$404
      w$=GETCMD$
      t$=LEFT$(w$,1)
      w$=MID$(w$,2,128)
      IF t$="X"
        endit:
      ELSEIF t$="C" OR t$="O"
        TRAP CLOSE
        IF ERR
          CLS :PRINT ERR$(ERR)
          GET :CONTINUE
        ENDIF
        fset:(t$)
      ENDIF
    ENDIF
  UNTIL a%(1)<256
  RETURN a%(1)
ENDP

PROC fset:(t$)
  LOCAL p%(6)
  IF t$="C"
    TRAP DELETE w$            REM SYS.SCREEN DOES ANY "OVERWRITE?"
    TRAP CREATE w$,A,A$
  ELSEIF t$="O"
    TRAP OPEN w$,A,A$
  ENDIF
  IF ERR
    CLS :PRINT ERR$(ERR)
    GET :STOP
```

```
  ENDIF
  SETNAME w$
ENDP


PROC endit:
  STOP
ENDP
```

You should, as in both these examples, be precise in checking for the System message; if in future the GETCMD$ function were to use values other than “C”, “O” or “X”, these procedures would ignore them.

If you need to check the modifier keys for the returned keypress, use `a%(2) AND $00ff` instead of KMOD.

SETNAME extracts the main part of the filename from any file specification (even one that is not DOS-like), in the same way as PARSE$. Using SETNAME ensures that the correct name will be used in the file list in the System screen. If an OPA lets you change files with its own ‘Open file’ option, it should always use SETNAME to inform the System screen of the new file in use.

To be strict, whenever **creating** a file, an OPA should first use PARSE$ to find the disk and directory requested. It should then use `TRAP MKDIR` to ensure that the directory exists.

When an OPA cannot respond

The LOCK command marks an OPA as locked or unlocked. When an OPA is locked with `LOCK ON`, the System will not send it events to change files or quit. If, for example, you move onto the file list in the System screen and press Delete to try to stop that running OPA, a message will appear, indicating that the OPA cannot close down at that moment.

You should use `LOCK ON` if your OPA uses a keyword, such as EDIT, which pauses the execution of the program. You might also use it when the OPA is about to go busy for a considerable length of time, or at any other point where a clean exit is not possible. Do not forget to use `LOCK OFF` as soon as possible afterwards.

An OPA is initially unlocked.

Designing an icon

As discussed earlier, an OPA icon is black and grey and has size 48 by 48 pixels. The icon is stored as two 48x48 bitmaps, black followed by grey, in a bitmap file. Here is a simple example program which creates a suitable bitmap:

```
PROC myicon:
  gCREATE(0,0,48,48,1,1)
  gBORDER $200
  gAT 6,28
  gPRINT "me!"
  gSAVEBIT "me"
ENDP
```

Here the window is created with a grey plane (the sixth argument to gCREATE) gSAVEBIT automatically saves a window with both black and grey plane to a file in the required format.

In the OPA itself use the ICON keyword, as explained previously, to give the name of the bitmap file to use here, `ICON "\opd\me"`.

OPAs and the status window

If you use `STATUSWIN ON,2` to display the status window, it shows the OPA's own icon and the name used with the APP keyword. `STATUSWIN ON,1` displays the smaller status window.

Important: The permanent status window is **behind** all other OPL windows. In order to see it, you must use FONT (or both SCREEN and gSETWIN) to reduce the size of the text and graphics windows. You should ensure that your program does not create windows over the top of it.

You can also display a list of modes/views for use with the ◆ key with DIAMINIT and position the ◆ indicator with `DIAMPOS`.

The name can be changed with the SETNAME command. In general, an OPA should use SETNAME whenever it changes files, or creates a new file.

Other TYPE options

You can add any of these numbers to the value you use with TYPE:

- $8000 (-32768) stops the System screen's 'New file' option from working, as for the RunOpl icon (translated OPL modules).
- $4000 (16384) stops the System screen from closing the OPA, as for the Time icon. You should not use this without a **very** good reason.
- $100 (8192) causes the System screen to terminate the OPA (when Delete is pressed there) without sending a message to the OPA to quit ("X"), as for the RunOpl icon again. This should be used only for OPAs which have no data that could be lost by sudden termination.

For example, use `TYPE $8001` for a type 1 OPA having the first of the features above. (Note that `TYPE $8000+1` would fail to translate as the translator cannot evaluate expressions for any keywords between APP and ENDA).

TRICKS

Ì The calculator memories

The calculator memories M0 to M9 are available as floating-point variables in OPL. You might use them to allow OPL access to results from the calculator, particularly if you use OPL procedures from within the calculator.

It's best not to use them as a substitute for declaring your own variables. Your OPL program might go wrong if another running OPL program uses them, or if you use them yourself in the calculator.

Running a program twice

Although you may never need to, you **can** run more than one copy of the same translated OPL module **at the same time**. There are two ways:

- Use 'Copy file' in the System screen to make a new copy of the module, with a different filename. Then run both files.
- **For the Series 3c only**, run the file as normal. Then move the highlight to under the RunOpl icon, press Tab to show the file selector, and pick the name of the translated module again.

Î Foreground and background

- `SETFLAGS $10000` tells the Series 5 to send a "machine switch on" event to the current program, whenever the Series 5 switches on, even if this program is in the background. If required, use it just once at the start of your program.
- The System OPX procedure `SETFOREGROUND:` brings the current program to the foreground.
- The System OPX procedure `SETBACKGROUND:` sends it to the background again.

Note that each of these should be followed by `gUPDATE` to ensure they take effect immediately.

Note that when a program runs in the background it can stop the "automatic turn off" feature from working. However, as soon as the program waits for a keypress or an event, with GET/GET$ or GETEVENT32, auto-turn off can occur.

Auto-turn off can also occur if the program does a PAUSE (of 2 or more 20ths of a second), but only if the program has used SETACTIVE from System OPX to mark the program as inactive.

See the 'Using OPXs on the Series 5' chapter for more information about System OPX procedures.

Foreground and background

- `CALL ($6c8d)` tells the Series 3c to send a "machine switch on" event to the current program, whenever the Series 3c switches on, even if this program is in the background. If required, use it just once at the start of your program.
- `CALL($198d,0,0)` brings the current program to the foreground.
- `CALL($198d,100,0)` sends it to the background again.

Each of these should be followed by `gUPDATE` to ensure they take effect immediately.

This example program comes to the foreground and beeps whenever you turn the Series 3c on. Be careful to enter the CALL and GETEVENT statements exactly as shown.

```
PROC beepon:
  LOCAL a%(6)
  PRINT "Hello"
  CALL($6c8d) :GUPDATE
  WHILE 1
  DO
  GETEVENT a%()
  IF a%(1)=$404 :STOP :ENDIF :REM closedown
  UNTIL a%(1)=$403 :REM machine ON
  CALL($198d,0,0) :gUPDATE
  BEEP 5,300 :PAUSE 10 :BEEP 5,500
  CALL($198d,100,0) :gUPDATE
  ENDWH
ENDP
```

Note that when a program runs in the background it can stop the "automatic turn off" feature from working. However, as soon as the program waits for a keypress or an event, with GET/GET$ or GETEVENT, auto-turn off can occur.

Auto-turn off can also occur if the program does a PAUSE (of 2 or more 20ths of a second), but only if the program has used `CALL($138b)` ("unmark as active")

On the Series 5, procedures are automatically cached and the cache commands are not available on the Series 5. Therefore, the following section is only applicable to the Series 3c and Siena.

Cacheing procedures on the Series 3c and Siena

Without procedure cacheing, procedures are loaded from file whenever they are called and discarded when they return - LOADM simply loads a map listing procedure names and their positions in the

module file so that they can be loaded fairly efficiently. The cache handling commands provide a method for keeping the code for a procedure loaded after it returns - it remains loaded until a procedure called later requires the space in the cache. The strategy is then to remove the least recently used procedures, making it more likely that all the procedures called together in a loop, for example, remain in the cache together, thus speeding up procedure calling significantly.

Cache handling keywords allow you to:

- create a cache of a specified initial and maximum size using `CACHE init%,max%`. You can specify these up to 32,767 bytes.

If you use hex, you can even exceed this figure, if you need to - e.g. `CACHE $9000,$9000`. However, you cannot exceed the 64k total memory limit which each Series 3c process has.

  - prevent loading and removal of procedures from the cache so that a given set of procedures can be guaranteed to remain in the cache using `CACHE OFF`. Procedures already in the cache are still used when cacheing is off. The loading and removal of procedures can subsequently be resumed using `CACHE ON`.
  - tidy the cache by removing procedures that are no longer in use (i.e. procedures that have returned) using `CACHETIDY`.
- for advanced use during program development, further keywords are provided for inspecting the contents of the cache at any time (see CACHEHDR and CACHEREC).

Cache size

Cacheing procedures is not a cure all. Care should be taken that the cache size is sufficient to load all procedures required for a fast loop otherwise, for example, a large procedure may cause all the small ones in a loop to be removed and equally, a small one may require the large one to be removed, so that the cache provides no benefit at all. In fact, the overhead needed for cache management can then make your program less efficient than having no cache at all. If the maximum cache size you can have is limited, careful use of `CACHE OFF` should prevent such problems at the expense of not fitting all the procedures in the loop in the cache. `CACHE OFF` is implemented very efficiently and calling it frequently in a loop should not cause much concern.

To guarantee that there is enough memory for a given cache size, create the cache passing that value as the initial size using `TRAP CACHE init%,max%`. TRAP ensures that if the cache creation succeeds, ERR returns zero and otherwise the negative 'Out of memory' error is raised. After creation, the cache will grow as required up to the maximum size `max%` or until there is not enough free memory to grow it. On failure to grow the cache, any procedures which will not fit into the existing cache, even when unused procedures are removed, are simply loaded without using the cache and are discarded when they return.

If you want to ensure a certain minimum cache size, say 10000 bytes, but do not care how large it grows, you could use `TRAP CACHE 10000,$ffff` so that the cache just grows up to the limits of memory. For a relatively small program, you might want to load the whole program into cache by making the cache size the same size as the module. This will in fact be a little larger than required, unnecessarily including a procedure name table and module file header which are not loaded into the cache. The minimum cache size is 2000 bytes, which is used if any lower value is specified. If the maximum size specified is less than the initial size, the maximum is set to the initial size. The maximum cache size cannot be changed once the cache has been created and an error is returned if you attempt to do so.

The initial cache size should ideally be large enough to hold all procedures that are to be cached simultaneously. There is no advantage in growing the cache from its initial size when you know that a certain minimum size is needed.

Procedures in unloaded modules

When a module is unloaded, all procedures in it that are no longer in use are removed from the cache. Any procedure that is still in use, is hidden in the cache by changing its first character to lower case; when it finally returns, a hidden procedure is removed in the normal manner to make room for loading

a new procedure when the cache is full. Note that it considered bad practice to unload a module containing procedures that are still running - e.g. for a procedure to unload its own module.

Cache timings

Calling an empty procedure that simply returns is approximately 10 times faster with a cache. This figure was obtained by calling such a procedure 10000 times in a loop, both with cacheing off and on, and subtracting the time taken running an empty loop in each case.

Clearly that case is one of the best for showing off the advantages of cacheing, and there is no general formula for calculating the speed gain. The procedures that benefit most will be those that need most module file access relative to their size in order to load them into memory. The programmer cannot reasonably write code taking this into account, so no further details are provided here.

The case described above does not require any procedures to be removed from the cache to make room for new procedures when the cache is full, and removal of procedures requires a fair amount of processing by the cache manager. If many procedures in a time-critical section of your program are loaded into the cache and not used often before removal, the speed gain may be less than expected - a larger cache may be called for to prevent too many removals.

It should be noted however, that even with the worst case of procedures being loaded into the cache for use just once before removal, having a cache is often superior to having no cache. This is because the cache manager reads module file data (required for loading the procedures into memory) in one block rather than a few bytes at a time and it is the avoidance of excessive file access which provides the primary speed gains for cacheing.

Compatibility mode modules

Procedures in modules translated for the Series 3 cannot be loaded into the cache. On encountering such a procedure, the cache manager simply loads it without using the cache and discards it when it returns. The reason for this is that a few extra bytes of data are stored in the Series 3c modules which are needed by the cache manager.

Potential problems in existing programs

It is possible that previously undiscovered bugs in existing OPL programs are brought to light simply by adding code to use the cache.

Without cacheing, the variables in a procedure are followed immediately by the code for the procedure. Writing beyond the variables (for example reading too many bytes into the final variable using such keywords as gPEEKLINE or KEYA) would have written over the code itself but would have gone unnoticed unless you happened to loop back to the corrupted code. With a cached procedure, the code no longer follows your variables, so the corruption occurs elsewhere in memory, resulting quite probably in the program crashing.

Controlling procedure cacheing

`TRAP CACHE initSize%,maxSize%` creates a cache of a specified initial number of bytes, which may grow up to the specified maximum. If the maximum is less than the initial size, the initial size becomes the maximum. If growing the cache fails, normal loading without the cache is used. The ‘In use’ error (-9) is raised if a cache has been created previously or the ‘Out of memory’ error (-10) on failure to create a cache of the specified initial size - use the TRAP command if required. Procedure code and other information needed for setting up variables are loaded into the cache when the procedure is called. If there is no space in the cache and enough space can be regained, the least recently used procedures are removed. Otherwise the procedure is loaded in the normal way without cacheing.

Once a cache has been created, `CACHE OFF` prevents further cacheing, although the cache is still searched when calling subsequent procedures. `CACHE ON` may then be used to re-enable cacheing. Note that `CACHE ON` or `CACHE OFF` are ignored if used before `CACHE initSize%,maxSize%`.

Tidying the cache

CACHETIDY removes any procedures from the cache that have returned to their callers. This might be called after performing a large, self-contained action in the program which required many procedures. Using CACHETIDY will then result in speedier searching for procedures called subsequently. More importantly, it will prevent the procedures being unloaded one at a time when the need arises - it is very efficient to remove a set of procedures that are contiguous in the cache as is likely to be the case in this situation.

Note that a procedure which has returned is automatically removed from the cache if you unload the module it is in, so CACHETIDY needn't be used for such a procedure.

Getting cache index header information

The CACHEHDR command is provided for advanced use and is intended for use during program development only.

`CACHEHDR ADDR(hdr%())` reads the current cache index header into the array `hdr%()` which must have at least 11 integer elements. Note that any information returned is liable to change whenever a procedure is called, so you cannot save these values over a procedure call.

If no cache has yet been created, `hdr%(10)=0` and the other data read is meaningless. Otherwise, the data read is as follows:

`hdr%(1)`	current address of the cache itself
`hdr%(2)`	number of procedures currently cached
`hdr%(3)`	maximum size of the cache in bytes
`hdr%(4)`	current size of the cache in bytes
`hdr%(5)`	number of free bytes in the cache
`hdr%(6)`	total number of bytes in cached procedures which are freeable (i.e. not running)
`hdr%(7)`	offset from the start of the cache index to the first free index record
`hdr%(8)`	offset from start of cache index to most recently used procedure's record; zero if none
`hdr%(9)`	offset from start of cache index to least recently used procedure's record; zero if none
`hdr%(10)`	address of the cache index, or zero if no cache created yet
`hdr%(11)`	non-zero if cacheing is on, and zero if it is off

The cache manager maintains an index for the cache consisting of an index header containing overall information for the whole cache as well as one index record for each procedure cached. All offsets mentioned above give the number of bytes from the start of the index to the procedure record specified. The index records for cached procedures form a doubly linked list, with one list beginning with the most recently used procedure (MRU), with offset given by `hdr%(8)`, and the other with the least recently used procedure (LRU) with offset given by `hdr%(9)`. A further singly linked list gives the offsets to free index records. The linkage mechanism is described in the discussion of CACHEREC below.

Getting a cache index record

The CACHEREC command is provided for advanced use and is intended for use during program development only.

`CACHEREC ADDR(rec%()),offset%` reads the cache index record (see the description of CACHEHDR above) at `offset%` into array `rec%()` which must have at least 18 integer elements. `offset%=0` specifies the most recently used (MRU) procedure's record if any and `offset%<0` the least recently used procedure (LRU) procedure's record if any.

The data returned by CACHEREC is meaningless if no cache exists (in which case `rec%(17)=0`) **or if there are no procedures cached yet** (when `hdr%(8)=0` as returned by CACHEHDR).

Each record gives the offset to both the more recently used and to the less recently used procedure's record in the linked lists, except for the MRU and the LRU procedures' records themselves which each terminate one of the lists with a zero offset. The first free index record (see CACHEHDR above) starts the free record list, in which each record gives the offset of the next free record or zero offset to terminate the list. To "walk" the cache index, you would always start by calling CACHEREC specifying either the MRU or LRU record offset, and use the values returned to read the less or more recently used procedure's record respectively. Note that any information returned is liable to change whenever a procedure is called, so you cannot save these values over a procedure call.

For the free cell list, only `rec%(1)` is significant, giving the offset of the next free index record. For the records in the lists starting with either the LRU or MRU record, the data returned in `rec%()` is:

`rec%(1)`	offset to less recently used procedure's record or zero if on LRU
`rec%(2)`	offset to more recently used procedure's record or zero if on MRU
`rec%(3)`	usage count zero if not running
`rec%(4)`	offset in cache itself to descriptor for building the procedure frame
`rec%(5)`	offset in cache itself to translated code for the procedure
`rec%(6)`	offset in cache itself to the end of the translated code for the procedure
`rec%(7)`	number of bytes used by the procedure in the cache itself
`rec%(8-15)`	leading byte counted procedure name, followed by some private data
`rec%(16)`	address of the procedure's leading byte counted module name
`rec%(17)`	address of the cache index, or zero if no cache created yet
`rec%(18)`	non-zero if cacheing is on, and zero if it is off

For example, to print the names of procedures and their sizes from MRU to LRU:

```
CACHEHDR ADDR(hdr%())
IF hdr%(10)=0
  PRINT "No cache created yet"
  RETURN
ENDIF
IF hdr%(8)=0                              REM MRU zero?
  PRINT "None cached currently"
  RETURN
ENDIF
rec%(1)=0                                 REM MRU first
DO
  CACHEREC ADDR(rec%()),rec%(1)           REM less recently used proc
  PRINT PEEK$(ADDR(rec%(8))),rec%(7)      REM name and size
UNTIL rec%(1)=0
```

Î **For Sprite handling on the Series 5, see the 'Using OPXs on the Series 5' chapter.** Note, however, that the basic idea of sprite handling remains the same for the Series 5, and you may find some of the information given below helpful.

Sprite handling on the Series 3c and Siena

How sprites work

OPL includes a set of keywords for handling a *sprite* - a user-defined black/grey/white graphics object of variable size, displayed on the screen at a specified position.

The sprite can also be *animated* - you can specify up to 13 *bitmap-sets* which are automatically presented in a cycle, with the duration for each bitmap-set specified by you. Each bitmap-set may be displayed at a specifiable offset from the sprite's notional position.

The 13 bitmap-sets are each composed of up to six bitmaps. The set pixels in each bitmap specify one of the following six actions: black pixels to be drawn; black pixels to be cleared; black pixels to be inverted; grey pixels to be drawn; grey pixels to be cleared; or grey pixels to be inverted. The bitmaps in a set must have the same size.

All the bitmaps in a set are drawn to the screen together and displayed for the specified duration, followed by the next set, and so on.

If you do not specify that a pixel is to be drawn, cleared or inverted, the background pixel is left unchanged.

Black pixels are drawn "on top of" grey pixels, so if you clear/invert just the grey pixels in the sprite they will be hidden under any pixels set black. So to clear/invert pixels on a background which has both grey and black pixels set, you need to clear/invert both black and grey pixels in the sprite.

The pixels of one colour (black or grey) which are set in one bitmap of the bitmap-set should not overlap with those of the same colour which are set in another bitmap in the same bitmap-set. This is because the order in which the bitmaps are applied is undefined. So, for example, do not specify that pixel (0,0) should have the black pixel both drawn and cleared.

Why use sprites?

A sprite is useful for displaying something in foreground without having to worry about restoring the background display. A sprite can also have any shape, leaving the background display all around it intact, and it can even be hollow - only the pixels specified by you are drawn, cleared or inverted. Typically only one bitmap-set containing two black bitmaps would be used - one for setting and one for clearing pixels.

You would not often use the sprite features in their full generality. In fact, more than one bitmap-set is needed only for animation and it is also seldom necessary to use all the available bitmaps in a single bitmap-set.

Creating a sprite

`sprId%=CREATESPRITE` creates a sprite and returns the sprite ID.

Appending a bitmap-set to a sprite

```
APPENDSPRITE tenths%,bitmap$()
```

```
APPENDSPRITE tenths%,bitmap$(),dx%,dy%
```

append a single bitmap-set to a sprite. These may be called up to 13 times for each sprite. APPENDSPRITE may be called only before the sprite is drawn, otherwise it raises an error. `tenths%` gives the duration in tenths of seconds for the bitmap-set to be displayed before going on to the next bitmap-set in the sequence. It is ignored if there is only one bitmap-set.

`bitmap$()` contains the names of the six bitmap files in the set:

`bitmap$(1)` for setting black pixels

`bitmap$(2)` for clearing black pixels

`bitmap$(3)` for inverting black pixels

`bitmap$(4)` for setting grey pixels

`bitmap$(5)` for clearing grey pixels

`bitmap$(6)` for inverting grey pixels

Use "" to specify no bitmap. If "" is used for all the bitmaps in the set, the sprite is left blank for the specified duration.

The array must have at least 6 elements.

All the bitmaps in a single bitmap-set must be the same size, otherwise an 'Invalid arguments' error is raised on attempting to draw the sprite. Bitmaps in different bitmap-sets may differ in size. `dx%` and `dy%` are the (x,y) offsets from the sprite position (see CREATESPRITE) to the top-left of the bitmap-set with positive for right and down. The default value of each is zero.

Sprites may use considerable amounts of memory. A sprite should generally be created, initialised and closed in the same procedure to prevent memory fragmentation. Care should also be taken in error handling to close a sprite that is no longer in use.

Creating or changing a sprite consisting of many bitmaps requires a lot of file access and should therefore be avoided if very fast sprite creation is required. Once the sprite has been drawn, no further file access is performed (even when it is animated) so the number of bitmaps is no longer important.

Drawing a sprite

`DRAWSPRITE"x%,y%` draws a sprite in the current window with top-left at pixel position `(x%,y%)`. The sprite must previously have been initialised using APPENDSPRITE or the 'Resource not open' error (-15) is raised. If any bitmap-set contains bitmaps with different sizes, DRAWSPRITE raises an 'Invalid arguments' error (-2).

Changing a bitmap-set in a sprite

```
CHANGESPRITE index%,tenths%,var bitmap$()
```

```
CHANGESPRITE index%,tenths%,var bitmap$(),dx%,dy%
```

change the bitmap-set specified by `index%` (1 for the first bitmap-set) in the sprite using the supplied bitmap files, offsets and duration which are all used in the same way as for APPENDSPRITE.

CHANGESPRITE can be called only after DRAWSPRITE.

Note that if all or many bitmap-sets in the sprite need changing or if each bitmap-set consists of many bitmaps, the time required to read the bitmaps from file may be considerable, especially if fast animation is in progress. In such circumstances, you should think about closing the sprite and creating a new one, which will often be more efficient.

Positioning a sprite

`POSSPRITE x%,y%` sets the position of the sprite to `(x%,y%)`.

Closing a sprite

`CLOSESPRITE sprId%` closes the sprite with ID `sprId%`.

Sprite example

The following code illustrates all the sprite handling keywords using a sprite consisting of just two bitmap-sets each containing a single bitmap.

```
PROC sprite:
  LOCAL bit$(6,6),sprId%
  crBits:                          REM create bitmap files
  gAT gWIDTH/2,0
  gFILL gWIDTH/2,gHEIGHT,0         REM fill half of screen
```

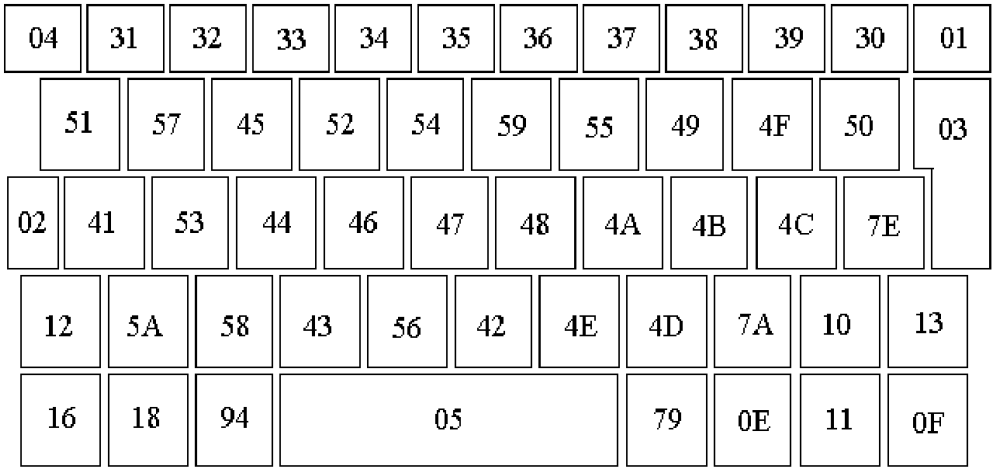
```
  sprId%=CREATESPRITE bit$(1)="" :bit$(2)=""
  bit$(3)="cross"                   REM black cross, pixels inverted
  bit$(4)="" :bit$(5)="" :bit$(6)="
  APPENDSPRITE 5,bit$(),0,0         REM cross for half a second
  bit$(1)="" :bit$(2)="" :bit$(3)=""
  bit$(4)="" :bit$(5)="" :bit$(6)=""
  APPENDSPRITE 5,bit$(),0,0         REM blank for half a second
  DRAWSPRITE gWIDTH/2-5,gHEIGHT/2-5       REM animate the sprite
  BUSY "flash cross, c",3           REM no offset ('c' for central)
  GET bit$(3)="box"                 REM black box, pixels inverted
  CHANGESPRITE 2,5,bit$(),0,0       REM in 2nd bitmap-set
  BUSY "cross/box, c/c",3           REM central/central
  GET
  CHANGESPRITE 2,5,bit$(),40,0      REM offset by 40 pixels right
  BUSY "cross/box, c/40",3          REM central/40
  GET
  bit$(3)=""                        REM Remove the cross in set 1
  CHANGESPRITE 1,3,bit$(),0,0       REM display for 3/10 seconds
  BUSY "flash box, 40",3            REM box at offset 40 still
  GET
  bit$(3)="cross"
  CHANGESPRITE 1,5,bit$(),0,0       REM cross centralised - set 1
  bit$(3)="box"
  CHANGESPRITE 2,5,bit$(),0,0       REM box centralised - set 2
  BUSY "Escape quits"
  DO
    POSSPRITE RND*(gWIDTH-11),RND*(gHEIGHT-11
                                    REM move sprite randomly
    PAUSE -20                       REM once a second
  UNTIL KEY = 27
  CLOSESPRITE sprId%
ENDP

PROC crBits:
  REM create bitmap files if they don't exist
  IF NOT EXIST("cross.pic") OR NOT EXIST("box.pic")
    gCREATE(0,0,11,11,1,1)
    gAT 5,0 :gLINEBY 0,11
```

```
    gAT 0,5 :gLINEBY 11,0
    gSAVEBIT "cross"
    gCLS
    gAT 0,0
    gBOX gWIDTH,gHEIGHT
    gSAVEBIT "box"
    gCLOSE gIDENTITY
  ENDIF
ENDP
```

SCANNING THE KEYBOARD DIRECTLY

It is sometimes useful to know which keys are being pressed at a given moment and also when a key is released. For example, in a game, a certain key might start some action and releasing the key might stop it.

Î `GETEVENT32 ev&()` and `GETEVENTA32 ev&()` return the scan code of any key pressed to the third element of the array `ev&()`. It is the programmers responsibility to track to key currently being pressed. The scan codes are given in hex in the following representation of the keyboard:



Ì `CALL($288e,ADDR(scan%()))` returns with the array `scan%()`, which must have at least 10 elements, containing a bit set for keys currently being pressed.

Every key on the keyboard is represented by a unique bit. This includes the modifier keys (Shift, Control etc.) and the application buttons (System, Data, Word etc.).

A set bit simply signifies a pressed key - a key pressed on its own gives one bit set; that same key with a modifier gives the same bit set with another bit for the modifier; the modifier on its own gives the same modifier bit on its own.

The following table lists each key (according to the text printed on the physical key itself), the `scan%()` array element for that key and the hexadecimal bit mask to be ANDed with that array element to check whether the key is being pressed.

key	`scan%()`	*mask*	*key*	`scan%()`	*mask*
System	5	$200	Data	4	$200

Word	6	$200	Agenda	2	$200
Time	1	$200	World	3	$200
Calc	2	$100	Sheet	1	$100
Esc	8	$100	1	8	$02
2	8	$04	3	6	$40
4	5	$04	5	5	$08
6	8	$08	7	4	$40
8	3	$08	9	3	$10
0	2	$10	+	2	$08
Delete	3	$01	Tab	1	$04
Q	7	$02	W	7	$20
E	6	$20	R	5	$02
T	5	$10	Y	1	$08
U	4	$20	I	3	$04
O	3	$20	P	2	$20
-	2	$04	Enter	1	$01
Control	3	$80	A	7	$04
S	7	$10	D	6	$10
F	6	$02	G	5	$20
H	8	$40	J	4	$10
K	3	$02	L	3	$40
*	2	$40	/	2	$02
Left shift	2	$80	Z	7	$08
X	7	$40	C	6	$08
V	6	$04	B	5	$40
N	1	$40	M	4	$08
,	4	$02	.	8	$10
Up	8	$20	Right shift	4	$80
Psion	1	$80	Menu	6	$80
	5	$80	Space	5	$01
Help	4	$04	Left	1	$10
Down	1	$20	Right	1	$02

For example, pressing Tab sets bit 2 of `scan%(1)`, pressing Control sets bit 7 of `scan%(3)` and pressing both together sets both these bits. So Tab is being pressed if `scan%(1) AND $04` is non-zero, and Control is being pressed if `scan%(3) AND $80` is non-zero.

A possible strategy for scanning the keys might be to wait for any key of interest using GETEVENT or GET (allowing switch off and less intensive use of the battery), start the required action, which is to be continued only while the key is being pressed, scan the keyboard, as discussed above, until the key is released, and then stop the action, wait for the next key and repeat.

Note that the key returned by GETEVENT or GET is not precisely synchronised with those scanned, so once you have waited for a relevant key you should scan for all the keys pressed, ignoring the keycode returned by GETEVENT or GET.

I/O FUNCTIONS AND COMMANDS

Note that one of the fundamental differences between the Series 3c and the Series 5 is that while any OPL program on the Series 3c has a 64K limit on the memory it may use, the memory which an OPL program may use on the Series 5 is unlimited up to the constraint placed by the machine itself. Therefore while 16-bit addresses are sufficient on the Series 3c, the Series 5 requires 32-bit addressing. As far as I/O functions are concerned, this means that any argument which is an address is an integer on the Series 3c, while it must be a long integer on the Series 5. See also the later section on '32-bit addressing'.

OPL includes powerful facilities to handle input and output ('I/O'). These functions and commands can be used to access all types of files on the Psion, as well as various other parts of the low-level software.

This section describes how to open, close, read and write to files, and how to set the position in a file. The data file handling commands and functions have been designed for use specifically with data files. **The I/O functions and commands are designed for general file access.** You don't need to use them to handle data files.

These are powerful functions and commands and they must be used with care. Before using them you must read this chapter closely and have a good grounding in OPL in general.

Error handling

You should have a good understanding of error handling before using the I/O functions.

The functions in this section never raise an OPL error message. Instead they return a value - if this is less than zero an error has occurred. It is the responsibility of the programmer to check all return values, and handle errors appropriately. Any error number returned will be one of those in the list given in the error handling chapter. You can use ERR$ to display the error as usual.

Handles

Many of these functions use a *handle*, which must be a long integer variable on the Series 5 and an integer variable on the Series 3c (see the note above). IOOPEN assigns this handle variable a value, which subsequent I/O functions use to access that particular file. Each file you IOOPEN needs a **different** handle variable.

var variables

In this section, `var` denotes an argument which should normally be a LOCAL or GLOBAL variable. (Single elements of arrays may also be used, but not field variables or procedure parameters.) Where you see `var` the **address** of the variable is passed, not the value in it. (This happens **automatically**; don't use ADDR yourself.)

In many cases the function you are calling passes information back by setting these `var` variables.

`var` is just to show you where you must use a suitable variable; you don't actually type it.

Ì For example: `ret%=IOOPEN(var handle%,name$,mode%)` in the syntax description indicates that `IOOPEN(h%,"abc",0)` is correct (provided `h%` is a simple variable), but `IOOPEN(100,"abc",0)` is incorrect.

Î For example: `ret%=IOOPEN(var handle&,name$,mode%)` in the syntax description indicates that `IOOPEN(h&,"abc",0)` is correct (provided `h&` is a simple variable), but `IOOPEN(100,"abc",0)` is incorrect.

It is possible, though, that you already have the address of the variable to use. It might be that this address is held in a field variable, or is even a constant value, but the most common situation is when the address was passed as a parameter to the current procedure.

If you add a # prefix to a var argument, this tells OPL that the expression following is the address to be used, not a variable whose address is to be taken.

Here is an example program:

```
PROC doopen:(phandle&, name$, mode%)
  LOCAL error%
  REM IOOPEN, handling errors
  error% = IOOPEN(#phandle&, name$, mode%)
  IF error% : RAISE error% : ENDIF
ENDP
```

The current value held in phandle& is passed to IOOPEN. You might call doopen: like this:

```
local filhand%,...
...
doopen:(addr(filhand%),"log.txt",$23)
...
```

The doopen: procedure calls IOOPEN with the address of filhand%, and IOOPEN will write the handle into filhand%.

Ì Note that phandle& should be replaced by phandle% (i.e. integer rather than long integer) on the Series 3c.

If you ever need to add or subtract numbers from the address of a variable, use the UADD and USUB functions, or you run the risk of 'Overflow' errors.

Opening a file with IOOPEN

```
ret%=IOOPEN(var handle%,name$,mode%
```

or

Î `ret%=IOOPEN(var handle%,address&,mode%)`

Ì `ret%=IOOPEN(var handle%,address%,mode%)`

for unique file creation.

Creates or opens a file (or device) called name$ and sets handle% to the handle to be used by the other I/O functions.

mode% specifies how the file is to be opened. It is formed by ORing together values which fall into the three following categories:

Mode Category 1 - Open mode

One and only one of the following values must be chosen from this category.

$0000 Open an existing file (or device). The initial current position is set to the start of the file.

$0001 Create a file which must not already exist.

$0002 Replace a file (truncate it to zero length) or create it if it does not exist.

$0003 Open an existing file for appending. The initial current position is set to the end of the file. For text format files (see $0020 below) this is the only way to position to end of file.

$0004 Creates a file with a unique name. For this case, you must use the address of a string instead of name$. This string specifies only the path of the file to be created (any file name in the string is ignored). The string at address% is then set by IOOPEN to the unique file name generated

(this will include the full path). The string must be large enough to take 130 characters (the maximum length file specification). For example:

```
s$="C:\home\"      REM C should be replaced with M on Series 3c
IOOPEN(handle%,ADDR(s$),mode%)
```

This mode is typically used for temporary files which will later be deleted or renamed.

Mode Category 2 - File format

One and only one of the following values must be chosen from this category. When creating a file, this value specifies the format of the new file. When opening an existing file, make sure you use the format with which it was created.

$0000 The file is treated as a byte stream of binary data with no restriction on the value of any byte and no structure imposed upon the data. Up to 16K can be read from or written to the file in a single operation.

$0020 The file is treated as a sequence of variable length records. The records are assumed to contain text terminated by any combination of the CR and LF (`$0D`, `$0A`) characters. The maximum record length is 256 bytes and Control-Z (`$1A`) marks the end of the file.

Î On the Series 5, all of these values are declared as constants in Const.oph. See the 'Calling Procedures' chapter for details of how to use this file and Appendix E for a listing of it.

Mode Category 3 - Access flags

Any combination of the following values may be chosen from this category.

$0100 Update flag. Allows the file to be written to as well as read. If not set, the file is opened for reading only. You **must** use this flag when creating or replacing a file.

$0200 Choose this value if you want the file to be open for *random* access (not sequential access), using the IOSEEK function.

$0400 Specifies that the file is being opened for sharing for example, with other running programs. Use if you want to read, not write to the file. If the file is opened for writing (`$0100` above), this flag is ignored, since sharing is then not feasible. If not specified, the file is locked and may only be used by this running program.

Closing a file with IOCLOSE

Files should be closed when no longer being accessed. This releases memory and other resources back to the system.

```
ret%=IOCLOSE(handle%)
```

Closes a file (or device) with the handle `handle%` as set by IOOPEN.

Reading a file with IOREAD

Î
```
ret%=IOREAD(handle%,address&,maxLen%)
```

Ì
```
ret%=IOREAD(handle%,address%,maxLen%)
```

Reads up to `maxLen%` bytes from a file with the handle `handle%` as set by IOOPEN. `address&` (or `address%` on the Series 3c) is the address of a buffer into which the data is read. This buffer must be large enough to hold a maximum of `maxLen%` bytes. The buffer could be an array or even a single integer as required. No more than 16K bytes can be read at a time.

The value returned to `ret%` is the actual number of bytes read or, if negative, is an error value.

Text files

If maxLen% exceeds the current record length, data only up to the end of the record is read into the buffer. No error is returned and the file position is set to the next record.

If a record is longer than maxLen%, the error value 'Record too large' (-43) is returned. In this case the data read is valid but is truncated to length maxLen%, and the file position is set to the next record.

A string array buffer$(255) could be used, but make sure that you pass the address ADDR(buffer$)+1 (UADD(ADDR(buffer$),1) on the Series 3c) to IOREAD. This leaves the leading byte free. You can then POKEB the leading byte with the count (returned to ret%) so that the string conforms to normal string format. See the example program.

Binary files

If you request more bytes than are left in the file, the number of bytes actually read (even zero) will be less than the number requested. So if ret%<maxLen%, end of file has been reached. No error is returned by IOREAD in this case, but the next IOREAD would return the error value 'End of file' (-36).

To read up to 16K bytes (8192 integers), you could declare an integer array buffer%(8192).

Writing to a file

Î `ret%=IOWRITE(handle%,address&,length%)`

Ì `ret%=IOWRITE(handle%,address%,length%)`

Writes length% bytes stored in a buffer at address& (address% on the Series 3c) to a file with the handle handle%.

When a file is opened as a binary file, the data written by IOWRITE overwrites data at the current position.

When a file is opened as a text file, IOWRITE writes a single record; the closing CR/LF is automatically added.

Positioning within a file

```
ret%=IOSEEK(handle%,mode%,var offset&)
```

Seeks to a position in a file that has been opened for random access (see IOOPEN above).

mode% specifies how the argument offset& is to be used. offset& may be positive to move forwards or negative to move backwards. The values you can use for mode% are:

1 Set position in a binary file to the absolute value specified in offset&, with 0 for the first byte in the file.

2 Set position in a binary file to offset& bytes from the end of the file

3 Set position in a binary file to offset& bytes relative to the current position.

6 Rewind a text file to the first record. offset& is not used, but you must still pass it as a argument, for compatibility with the other cases.

IOSEEK sets the variable offset& to the absolute position set.

Example - displaying a plain text file

This program opens a plain text file, such as one created with the 'Export as text…' option in the 'File' menu of the Program editor on the Series 5, or the 'Save as' option in the Word Processor or Database applications on the Series 3c, and types it to the screen. Press Esc to quit and any other key to pause the typing to the screen.

```
PROC ioType:
```

```
  LOCAL ret%,fName$(128),txt$(255),address&
  LOCAL handle%,mode%,k%
  PRINT "Filename?", :INPUT fName$ : CLS
  mode%=$0400 OR $0020        REM open=$0000, text=$0020, share=$0400
  ret%=IOOPEN(handle%,fName$,mode%)
  IF ret%<0
    showErr:(ret%)
    RETURN
  ENDIF
  address&=ADDR(txt$)
  WHILE 1
    k%=KEY
    IF k%                     REM if keypress
      IF k%=27                REM Esc pressed
        RETURN                REM otherwise wait for a key
      ELSEIF GET=27
        RETURN                REM Esc pressed
      ENDIF
    ENDIF
    ret%=IOREAD(handle%,address&+1,255)
    IF ret%<0
      IF ret%<>-36            REM NOT EOF
        showErr:(ret%)
      ENDIF
      BREAK
    ELSE
      POKEB address&,ret%     REM leading byte count
      PRINT txt$
    ENDIF
  ENDWH
  ret%=IOCLOSE(handle%)
  IF ret%
    showErr:(ret%)
  ENDIF
  PAUSE -100 :KEY
ENDP

PROC showErr:(val%)
```

```
   PRINT "Error",val%,err$(val%)
   GET
ENDP
```

Note that this example may be used for any of the machines. However, any address variables may be changed from long integers to integers if you are using the Series 3c or Siena.

Asynchronous requests and semaphores0

This section provides the general background necessary for understanding how EPOC I/O devices can be accessed by an OPL application program.

Asynchronous requests

Many operating system services are implemented in two steps:

1. make the service request (sometimes referred to as the *queuing* of a request)
2. wait for the requested operation to complete

In most cases, as well as providing functions for each step, the system provides a function containing both the above steps. Such functions are called *synchronous* because they automatically synchronise the requesting process by waiting until the operation has completed. The internal function that makes the request without waiting for completion is called an *asynchronous* function.

Examples of asynchronous request functions are:

IOC (or IOA)for requests on an open I/O channel

KEYAfor requests on the keyboard channel

Î GETEVENTA32for requests of events from the widow server

Î PLAYSOUNDA:(in System OPX) for requests to play sound files

The synchronous versions of the above functions are IOW , GET, GETEVENT32 and PLAYSOUND: respectively.

Applications use asynchronous requests in situations like the following:

1. make request A
2. make request B
3. wait for either of the requested operations to complete

Processes wait for the completion of asynchronous requests by waiting on their *I/O semaphore* where each request is associated with a *status word*.

The I/O semaphore

When a process is created, the system automatically creates an *I/O semaphore* on its behalf (a more accurately descriptive name would have been the *asynchronous request semaphore*). After making one or more asynchronous requests using IOC or IOA, a process calls IOWAIT to wait on the I/O semaphore for one of the requests to complete. A typical application spends most of its time waiting on its I/O semaphore. For example, an interactive application process that is waiting for user input is waiting on the I/O semaphore.

Semaphores are provided to synchronise cooperating processes (where, in this context, a process includes a hardware interrupt). In OPL semaphores are used for synchronising the completion of asynchronous requests.

The semaphores in both the EPOC16 and EPOC32 operating systems are counting semaphores, having a signed value that is incremented by calling IOSIGNAL and decremented by calling IOWAIT. A semaphore with a negative value implies that a process must wait for the completion of an event.

The process or the hardware interrupt handler that implements the requested operation sends a signal to the process (using a generalised version of IOSIGNAL directed to the required process) to indicate that the operation has completed. If one or more wait handlers have been installed (wait handlers are described below), they may process the signal and re-signal using IOSIGNAL. In some cases, it is convenient for the requestor to use IOSIGNAL to signal itself and subsequently to process that signal in a central call to IOWAIT.

Status words

Although the arguments to asynchronous request functions vary, they all take a so-called *status word* argument, which subsequently contains the status of the requested operation. On the Series 3c the status word is always a 16-bit integer, like `stat%`. On the Series 5, the status word is also a 16-bit integer, except when calling an asynchronous OPX procedure that specifically requires a 32-bit integer status word, such as PLAYSOUNDA: in System OPX.

All asynchronous requests exhibit the following behaviour:

1. While the request is pending, a 16-bit status word contains the negative -46.

   Î A 32-bit status word however contains &80000001. Const.oph provides constant definitions for these values. For details of how to use this file see the ‘Calling Procedures’ chapter and see Appendix E for a listing of it.

2. When the operation has completed, a value other than -46 is written to the status word. This value is zero or positive to indicate success, or a negative error number to indicate failure. For 16-bit status words, an OPL error value is used. See the ‘Error Handling’ chapter for a list of these.

   Î For 32-bit status words on the Series 5, an EPOC32 error value is used. The EPOC32 error values are listed in Appendix G.

3. The requesting process’s I/O semaphore is signalled (after the status word has been written).

Making a request while a previous request on the same status word is still pending will normally result in a *system panic* where the program is killed immediately, without being able to trap the error.

When there are multiple requests, each request is associated with a different status word. After returning from IOWAIT, the caller typically *polls* each status word until one is found that contains other than -46. That completion is then processed (which might include renewing the asynchronous request) and IOWAIT is called again to process the next completion.

Cancelling an asynchronous request

Most asynchronous request functions made using IOC (or IOA) can be cancelled. To cancel such requests use the IOCANCEL function.

For asynchronous requests made using a mechanism other than IOC (or IOA), a specific cancelling function must be used instead:

- Use KEYC to cancel a KEYA request.
- Î Use GETEVENTC to cancel a GETEVENTA32 request.
- Î Use STOPSOUND&: to cancel a PLAYSOUNDA: request.

The following general principles apply to all functions that cancel an asynchronous request:

- the cancel precipitates the completion of the operation (it does not stop the operation from completing).
- the cancel may or not be effective (that is, the operation may complete naturally before the cancel is processed).

- after a cancel, you must still process the completion of the asynchronous request (typically by immediately calling IOWAITSTAT or IOWAITSTAT32, described below, to "use up" the signal). An exception to this rule is that GETEVENTC and KEYC themselves wait for the cancellation signal, so IOWAITSTAT must not be used.

A 16-bit status word is set to -48 when a request has been effective.

Î A 32-bit status word is set to -3 (EPOC32's error code for 'Cancel error') when a request using a 32-bit status word is cancelled.

Waiting for a particular completion

When waiting for the completion of a *particular* asynchronous request, the wait on the I/O semaphore must be sure that it is not fooled into a premature return by the completion of any other pending asynchronous request. This is done by calling IOWAITSTAT which behaves in a similar way to IOWAIT except that it only returns when the associated status word is other than -46.

Î For a 32-bit status word request, IOWAITSTAT32 is used instead, again behaving in a similar way to IOWAIT except that it only returns when the associated 32-bit status word is other than &80000001.

In general, IOWAITSTAT (IOWAITSTAT32) is a safer option than IOWAIT to "use up" the signal resulting from the cancelled operation. If the cancel is not immediately effective and another completion causes IOWAIT to return, the program could continue and make another request before the cancelled operation completes (which would result in a process panic).

A first example using asynchronous I/O

In the following example, the opened asynchronous timer `timChan%` is used to construct a synchronous function which attempts to write the passed string to the opened serial channel `serChan%`. If it takes more that 5 seconds to complete the write, the procedure raises the 'inactivity' error -54. For simplicity it is assumed that there are no other outstanding events which could complete and that both requests started successfully. Thus it is certain that on return from the call to IOWAIT, one of the two asynchronous requests has completed.

```
PROC strToSer:(inStr$)
  LOCAL str$(255)             REM local copy of inStr$ needed for
  ADDR
  LOCAL len%,timStat%,serStat%,timeout%,ret%,pStr&


  str$=inStr$
  pStr&=ADDR(str$)+1 REM pointer to string skipping leading count
  byte
  len%=LEN(str$)
  IOC(serChan%,2,serStat%,#pStr&,len%)
                              REM request asynchronous serial write
  timeout%=50                       REM 5 second timeout
  IOC(timChan%,1,timStat%,timeout%) REM Relative timer - function 1
```

```
  IOWAIT
  IF serStat%=-46                  REM must have timed out
    IOCANCEL(serChan%)             REM cancel serial request
    IOWAITSTAT serStat%            REM use up the signal
    RAISE -54                      REM inactivity timeout
   ENDIF
   IOCANCEL(timChan%)              REM cancel timer request
   IOWAITSTAT timStat%             REM use up the signal
ENDP
```

Ì Wait handlers

Wait handlers are functions that handle the completion of asynchronous requests from within IOWAIT (or IOWAITSTAT). Active wait handler functions are called just before IOWAIT would have otherwise returned.

Many I/O devices install a *device wait handler* when the device is opened.

Wait handlers are only called when the process calls IOWAIT (or a function such as IOWAITSTAT that calls IOWAIT). While an application performs a computationally intensive task that takes an extended time, it should consider calling IOYIELD (which effectively calls IOSIGNAL followed by IOWAIT) to allow any installed wait handlers to be called. Application programs must never assume that no wait handlers have been installed.

Polling rather than waiting

In a multi-tasking operating system it is extremely anti-social to wait for an operation to complete by polling the status word in a tight loop rather than call IOWAIT (because the polling will “hog” the processor to no benefit). However, when there is useful work to be done between each poll, it can be appropriate to poll - for example, to check periodically for user input while performing an extended calculation.

When a poll detects a completed status word, it is still obligatory to “use up” the signal by calling IOWAIT (otherwise you will get a “stray signal” later).

Î *Polling on the Series5*

The status word will be set only when either IOWAIT or IOYIELD is called. For a 32-bit status word, call IOWAITSTAT32 instead.

For example:

```
...
IOC(handle%,func%,stat%,#pBuf&,len%)
DO
  calc:    REM perform part of some calculation until request
  complete
  IOYIELD  REM allow status word to be set
UNTIL stat%<>-46
...
```

Ì *Polling on the Series 3c and Siena*

If this approach is used, you should be aware that in some cases asynchronous requests are completed by a wait handler and it is necessary to call IOYIELD *before* each poll to give any wait handlers a chance to run. For example:

```
...
IOC(handle%,func%,stat%,#pBuf%,len%)
DO
  calc:    REM perform part of some calculation until request
  complete
  IOYIELD  REM allow handler to run
UNTIL stat%<>-46
...
```

This case occurs when making requests on a device driver that services hardware interrupts. For example, the serial port driver services the hardware interrupt generated by the receipt of a serial frame. A hardware interrupt handler cannot write directly to the data segment of the requesting process because that data segment may be moving when the interrupt occurs. Instead, the interrupt handler must write first to a fixed memory location (either in the operating system variables if it is an in-built driver or in a device segment if it is an external driver) and then signal the I/O semaphore of the requesting process. When the requesting process next calls IOWAIT (directly or indirectly through say IOYIELD), the device driver's wait handler is called to copy the data safely to the process data segment.

Device drivers that are implemented by a server process (for example, the window server) do not require a wait handler to complete operations.

I/O device handling and the asynchronous request functions

The following I/O functions provide access to devices. A full description is not within the scope of this manual, since these functions require **extensive** knowledge of the Series 5 or Series 3c operating systems and related programming techniques. The syntax and argument descriptions are provided here for completeness.

The previous section explains in detail the semantics of asynchronous I/O. In the descriptions of the asynchronous I/O functions below, a careful reading of that section is assumed.

```
ret%=IOW(handle%,func%,var arg1,var arg2)
```

The device driver opened with `handle%` (as returned by IOOPEN) performs the synchronous I/O function `func%` with the two further arguments. The size and structure of these two arguments is specified by the particular device driver's documentation.

```
IOC(handle%,func%,var stat%,var a1,var a2)
IOC(handle%,func%,var stat%,var a1)
IOC(handle%,func%,var stat%)
```

Make an I/O request with guaranteed completion. The device driver opened with `handle%` (as returned by IOOPEN) performs the asynchronous I/O function `func%` with up to two further arguments. The size and structure of these arguments is specified by the particular device driver's documentation.

As explained in detail in the previous section, *asynchronous* means that the IOC returns immediately, and the OPL program can carry on with other statements. `status%` will always be set to -46, which means that the function is still pending.

When, at some later time, the function completes, status% is automatically changed. (For this reason, status% should usually be global since if the program is still running, status% must be available when the request completes, or the program will probably crash). If status%>=0, the function completed without error. If status%<0, the function completed with error. The return values and error codes are specific to the device driver.

If an OPL program is ready to exit, it does not have to wait for any signals from pending IOC calls.

```
ret%=IOA(handle%,func%,var status%,var arg1,var arg2)
```

The device driver opened with handle% (as returned by IOOPEN) performs the asynchronous I/O function func% with two further arguments. The size and structure of these two arguments is specified by the particular device driver's documentation.

This has the same form as IOC, but the return value **cannot** be ignored. IOC is effectively the same as:

```
ret%=IOA(h%,f%,stat%,...)
IF ret%<0
  stat%=ret%
  IOSIGNAL
ENDIF
```

IOC allows you to assume that the request started successfully - any error is always given in the status word stat%. If there was an error, stat% contains the error code and the IOSIGNAL causes the next IOWAIT to return immediately as if the error occurred **after** completion. There is seldom a requirement to know whether an error occurred on starting a function, and IOC should therefore nearly always be used in preference to IOA.

```
IOWAIT
```

Wait for an asynchronous request (such as one requested by IOC, KEYA or GETEVENTA32) to complete. IOWAIT returns when **any** asynchronous I/O function completes. Check status% to find out which request has completed. Use IOWAITSTAT with the relevant status word to wait for a particular request to complete.

```
IOSIGNAL
```

Replace a signal of an I/O function's completion.

As shown in 'A simple example using asynchronous I/O' in the previous section, it is sometimes useful to construct a synchronous operation from two asynchronous operations by waiting for either to complete before returning. In that case it waited for either the serial write request or a timeout. As noted there, the example assumed that there were no other outstanding asynchronous requests. If there had been one or more asynchronous requests made before calling that procedure, such as a request to play a sound file, and if that request had completed before both the serial write and the timer request, the procedure would incorrectly assume that the timer had completed:

```
IOWAIT
IF serStat%=-46                  REM must have timed out
  IOCANCEL(serChan%)             REM cancel serial request
  IOWAITSTAT serStat%            REM use up the signal
  RAISE -54                      REM inactivity timeout
 ENDIF
 IOCANCEL(timChan%)              REM cancel timer request
```

```
   IOWAITSTAT timStat%              REM use up the signal
```

To deal with this situation correctly, the procedure should instead check that one of the particular two requests it knows about has completed and re-signal to 'put back' the signal consumed for the sound file completion:

```
PROC strToSer:(inStr$)
  LOCAL str$(255)       REM local copy of inStr$ needed for ADDR
  LOCAL len%,timStat%,serStat%,timeout%,ret%,pStr&
  LOCAL signals%        REM count of external signals
  LOCAL err%

  str$=inStr$
  pStr&=ADDR(str$)+1 REM ptr to string skipping leading count byte
  len%=LEN(str$)
  IOC(serChan%,2,serStat%,#pStr&,len%)
                              REM request asynchronous serial write
  timeout%=50                       REM 5 second timeout
  IOC(timChan%,1,timStat%,timeout%) REM relative timer - function 1
  WHILE 1                           REM forever loop
    IOWAIT                          REM wait for any completion
    IF timStat%<>-46                REM timed out
      IOCANCEL(serChan%)            REM cancel serial request
      IOWAITSTAT serStat%           REM use up the signal
      err%=-54                      REM inactivity timeout (raised
                                    REM below after re-signalling)
      BREAK                         REM stop waiting and re-signal
    ELSEIF serStat%<>-46            REM serial write complete
      IOCANCEL(timChan%)            REM cancel timer request
      IOWAITSTAT timStat%           REM use up the signal
      BREAK                         REM stop waiting and re-signal
    ELSE                            REM other unknown request
      signals%=signal%+1            REM count other signals
                                    REM loop again for next
    ENDIF
  ENDWH
  WHILE signals%>0                  REM now re-signal any consumed
                                    REM external signals
    IOSIGNAL
    signals%=signals%-1
  ENDWH
```

```
  IF err%
    RAISE err%
  ENDIF
ENDP
```

IOSIGNAL is called only after exiting the IOWAIT loop, otherwise the signal would cause the IOWAIT to return immediately.

`IOWAITSTAT var status%`

Wait for a particular asynchronous function, called with IOC, to complete.

Î `IOWAITSTAT32 var stat&`

Similar to IOWAITSTAT but takes a 32-bit status word. IOWAITSTAT32 should be called only when you need to wait for completion of a request made using a 32-bit status word when calling an asynchronous OPX procedure. `status&` will be &80000001 while the function is still pending, and on completion will be set to the appropriate EPOC32 error code, listed in Appendix G. See also the 'Using OPXs on the Series 5' chapter and the 'Alphabetic listing' chapter.

`IOYIELD`

Ensures that any asynchronous function is given a chance to run. Some devices are unable to perform an asynchronous request if an OPL program becomes computationally intensive, using no I/O (screen, keyboard etc.) at all. In such cases, the OPL program should use IOYIELD before checking its `status%` variable. This is **always** the case on the Series 5. IOYIELD is the equivalent of IOSIGNAL followed by IOWAIT - the IOWAIT returns immediately with the signal from IOSIGNAL, but the IOWAIT causes any asynchronous handlers to run.

`IOCANCEL(handle%)`

Cancels any outstanding asynchronous I/O request made using IOC (or IOA) on the specified channel, causing them to complete with the completion status word containing -48 ('I/O cancelled'). The return value is always zero and may be ignored.

The IOCANCEL function is harmless if no request is outstanding (e.g. if the function completed just before cancellation requested).

`GETEVENTA32(var status%,var event&())`

This is an asynchronous window server read function. You must declare a long integer array with at least 16 elements. If a window server event occurs, the information is returned in `event&()` as described under GETEVENT32 in the alphabetic listing.

`GETEVENTC(var status%)`

Cancels a GETEVENTA32. Note that IOWAITSTAT should **not** be called after GETEVENTC - OPL consumes the GETEVENTC signal.

`err%=KEYA(var status%,key%())`

This is an asynchronous keyboard read function. You must declare an integer array with two elements here, `key%(1)` and `key%(2)` to receive the keypress information. If a key is pressed, the information is returned in this way:

- `key%(1)` is assigned the character code of the key.
- The least significant byte of `key%(2)` takes the key modifier, in the same way as KMOD 2 for Shift down, 4 for Control down and so on. KMOD cannot be used with KEYA.

- The most significant byte of `key%(2)` takes the count of keys pressed (0 or 1).

KEYA needs an IOWAIT in the same way as IOC.

Some useful IOW functions

IOW has this specification:

```
ret%=IOW(handle%,func%,var arg1,var arg2)
```

Here are some uses:

```
LOCAL a%(6)

IOW(-2,8,a%(),a%())            REM 2nd a% is ignored
```

senses the current text window (as set by the most recent SCREEN command) and the text cursor position **ignoring** any values already in the array `a%()`. This gives the same information as SCREENINFO, which should be used in preference (see the 'Alphabetic Listing' chapter).

The first four elements are set to represent the offset of the current text window from the default text window. `a%(1)` is set to the `x`-offset of the top left corner of the current text window from the default text window's top left corner and `a%(2)` to the `y`-offset of the top left corner of current text window. Similarly `a%(3)` and `a%(4)` give the offset of bottom right corner of text window from the bottom right corner of the default text window. For example, if the most recent SCREEN command was `SCREEN 10,11,4,5` the first four elements of the array `a%()` would be set to (3,4,13,15). The `x` and `y` positions of the cursor **relative to the current text window** are written to `a%(5)` and `a%(6)` respectively. All positions and offsets take `0,0`, not `1,1`, as the point at the top left.

```
LOCAL i%,a%(6)

i%=2

a%(1)=x1% :a%(2)=y1%

a%(3)=x2% :a%(4)=y2%

IOW(-2,7,i%,a%())
```

clears a rectangle at `x1%,y1%` (top left), `x2%,y2%` (bottom right). If `y2%` is one greater than `y1%`, this will clear part or all of a line.

Example of IOW screen functions

The final two procedures in this module call the two IOW screen functions described beforehand. The rest of the module lets you select the function and values to use. It uses the technique used in the 'Friendlier interaction' chapter of handling menus and short-cut keys by calling procedures with string expressions.

Î

```
PROC iotest:
  GLOBAL x1%,x2%,y1%,y2%
  LOCAL i%,h$(2),a$(5)
  x1%=2 :y1%=2
  x2%=25 :y2%=5                    REM our test screensize
  SCREEN x2%-x1%,y2%-y1%,x1%,y1%
  AT 1,1
```

```
  PRINT "Text window IO test"
  PRINT "Control-Q quits"
  h$="cr"                           REM our short-cut keys
  DO
    mINIT
    mCARD "Set","Rect",%r
    mCARD "Sense","Cursor",%c
    i%=MENU
    IF i% AND INTF(LOC(h$,CHR$(i%)))
      a$="proc"+chr$(i%)
      @(a$):
    ENDIF
    ELSEIF i% AND $200              REM hot-key
      i%=(i%-$200)
      i%=LOC(h$,CHR$(i%))           REM One of ours?
      IF i%
        a$="proc"+MID$(h$,i%,1)
        @(a$):
      ENDIF                         REM ignore other weird keypresses
    ELSE                            REM some other key, so return it
      PRINT CHR$(i%);
    ENDIF
  UNTIL 0
ENDP

PROC procc:
  LOCAL a&
  a&=iocurs&:
  PRINT "x";1+(a& AND &ffff);
  PRINT "y";1+(a&/&10000)
ENDP

PROC procr:
  LOCAL xx1%,yy1%,xx2%,yy2%
  LOCAL xx1&,yy1&,xx2&,yy2&
  dINIT "Clear rectangle"
  dLONG xx1&,"Top left x",1,x2%-x1%
  dLONG yy1&,"Top left y",1,y2%-y1%
```

```
  dLONG xx2&,"Bottom left x",2,x2%-x1%
  dLONG yy2&,"Bottom left y",2,y2%-y1%
  IF DIALOG
    xx1%=xx1&-1 :xx2%=xx2&-1
    yy1%=yy1&-1 :yy2%=yy2&-1
    iorect:(xx1%,yy1%,xx2%,yy2%)
  ENDIF
ENDP

PROC iocurs&:
  LOCAL a%(4),a&
  REM don't change the order of these!
  a%(1)=x1% :a%(2)=y1%
  a%(3)=x2% :a%(4)=y2%
  IOW(-2,8,a%(),a%()) REM 2nd a% is ignored
  RETURN a&
ENDP

PROC iorect:(xx1%,yy1%,xx2%,yy2%)
  LOCAL i%,a%(6)
  i%=2 :REM "clear rect" option
  a%(1)=xx1% :a%(2)=yy1%
  a%(3)=xx2% :a%(4)=yy2%
  IOW(-2,7,i%,a%())
ENDP

PROC iotest:
  GLOBAL x1%,x2%,y1%,y2%
  LOCAL i%,h$(2),a$(5)
  x1%=2 :y1%=2
  x2%=25 :y2%=5                    REM our test screensize
  SCREEN x2%-x1%,y2%-y1%,x1%,y1%
  AT 1,1
  PRINT "Text window IO test"
  PRINT "Psion-Esc quits"
  h$="cr"                          REM our hot-keys
  DO
    i%=GET
```

```
    IF i%=$122                      REM Menu key
      mINIT
      mCARD "Set","Rect",%r
      mCARD "Sense","Cursor",%c
      i%=MENU
      IF i% AND INTF(LOC(h$,CHR$(i%)))
        a$="proc"+chr$(i%)
        @(a$):
      ENDIF
    ELSEIF i% AND $200              REM hot-key
      i%=(i%-$200)
      i%=LOC(h$,CHR$(i%))           REM One of ours?
      IF i%
        a$="proc"+MID$(h$,i%,1)
        @(a$):
      ENDIF                         REM ignore other weird keypresses
    ELSE                            REM some other key, so return it
      PRINT CHR$(i%);
    ENDIF
  UNTIL 0
ENDP
```

Î The following two examples could not be used on the Series 5. The use of sound in OPL on the Series 5 is handled with the use of System OPX. See the 'Using OPXs on the Series 5' chapter for further details of how to use this.

Ì Alarm example - IOC to ALM:

The `ALM:` device provides access to alarms. When writing to it with IOW, IOC or IOA, you can use these two functions:

- function=1 - only the date (and no time) is shown on the screen when the alarm rings - e.g. `Thu 5 Sep`
- function=2 - the day and time are shown - e.g. `Thu 11:54`

In either case, you must pass these two arguments:

- An array of 2 long integers the first is the time for the alarm to go off, and the second is the time for which it is due. Both are given in seconds since midnight on 1/1/1970.
- A message, as a **zero-terminated** string of up to 64 characters.

This procedure asks for the information for an alarm, and sets it (as type 2 day and time to be shown when the alarm rings). If you press the Time button, and this is the next alarm to ring, it is shown as a RunOpl alarm.

```
PROC alm:
  LOCAL h%,a&(2),a$(64),b$(65),d&,t&,t2&,a%,r%,s%
```

```
  r%=IOOPEN(h%,"ALM:",0)
  IF r%<0 :RAISE r% :ENDIF
  d&=DAYS(DAY,MONTH,YEAR)           REM today
  t&=DATETOSECS(1970,1,1,HOUR,MINUTE,0)
  dINIT "Set alarm"
  dTIME t&,"Time",0,0,DATETOSECS(1970,1,1,23,59,59)
  dDATE d&,"Date",d&,DAYS(31,12,2049)
  dTIME t2&,"Alarm advance time",2,0,86399
  dEDIT a$,"Message"
  IF DIALOG
    a&(2)=86400*(d&-25567)+t&
    a&(1)=a&(2)-t2&
    b$=a$+CHR$(0)                   REM zero-terminate the string
    IOC(h%,2,s%,a&(),#UADD(ADDR(b$),1))
  ENDIF
  IOCLOSE(h%)
ENDP
```

At the moment the alarm rings, either `s%` must still be available to take the status word set by this `ALM:` function, or the program must have exited. Otherwise the status word will be written to a random area of memory. So in this example, no error-checking is done after the IOC - the program just ends. So you would have to use this as a whole program itself - you must not call this procedure as written here from another procedure.

Ì Dialling example - IOW to SND:

The `SND:` device provides sound services on the Series 3c. One function, number 10, provides access to DTMF dialling. It requires these two arguments:

- The number to dial, as a zero-terminated string of up to 24 characters.
- An array of 2 integers. The first is the tone length (*256) plus the delay length, and the second is the pause length. All of these are specified in 1/32 s of a second.

```
PROC dtmf:
  LOCAL h%,a$(24),b$(25),z%,r%,a%(2)
  r%=IOOPEN(h%,"SND:",0)
  IF r%<0 :RAISE r% :ENDIF
  dINIT
  dEDIT a$,"Dial"
  IF DIALOG
    a%(1)=8+(256*8)
    a%(2)=48
    b$=a$+CHR$(0)
    r%=IOW(h%,10,#UADD(ADDR(b$),1),a%())
    IF r%<0 :RAISE r% :ENDIF
```

```
  ENDIF
  r%=IOCLOSE(h%)
  IF r%<0 :RAISE r% :ENDIF
ENDP
```

Î **The following section does not apply to the Series 5.** The use of sound in OPL on the Series 5 is handled with the use of System OPX. See the 'Using OPXs on the Series 5' chapter for further details of how to use this.

Recording and playing sounds on the Series 3c and Siena

This section explains how to write a program which records sounds to a *sound file* using the in-built Series 3c microphone, and which plays these or pre-recorded sound files back.

Sound file structure

Series 3c sound files are files with a `.WVE` extension that contain a 32-byte header and a byte stream of digital sound which is sampled and played back at 8000 bytes per second (12-bit sound is converted to 8-bit using A-Law encoding).

The file header has the following format:

offset in file	*bytes*	*contents*
0	16	zero-terminated 'ALawSoundFile**'
16	2	version of this format
18	4	number of 8-bit samples
22	2	trailing silence in ticks
24	2	repeats
26	6	spare bytes reserved for future use

The number of samples is the number of bytes following the header and should always be size of file less 32 for the header.

The silence in ticks is the number of system ticks of silence appended to each repeat on playback (in practice, you get at least 2 ticks between repeats). A system tick is $^{1}/_{32}$ seconds or 250 samples.

The repeats are the number of times to repeat the sound on playback (0 and 1 are treated as the same).

You can truncate the sound file, change the number of repeats and the trailing silence by changing the file length and header using the I/O binary file handling functions (IOOPEN, IOW, IOSEEK, IOWRITE etc.) described elsewhere in this chapter.

For example, you can truncate the file to length `newLen&` using:

```
ret%=IOW(handle%,11,newLen&,#0)
```

How to record and play sounds

The following set of procedures perform asynchronous recording and playing of sounds.

```
PROC recorda:(pstat%,inname$,size%)
  LOCAL name$(128)
  name$=inname$+chr$(0)
  CALL($2186,UADD(ADDR(name$),1),size%,0,0,pstat%)
ENDP
```

```
PROC recordc:
  CALL($2386)
ENDP

PROC recordw%:(inname$,size%)
  LOCAL name$(128),p%,ret%
  p%=PEEKW($1c)+6             REM address of saved flags after CALL
  name$=inname$+chr$(0)
  ret%=CALL($2286,UADD(ADDR(name$),1),size%)
  IF PEEKW(p%) AND 1          REM carry set for error
    RETURN ret% OR $FF00      REM return error
  ENDIF
ENDP

PROC playa:(pstat%,inname$,ticks%,vol%)
  LOCAL name$(128)
  name$=inname$+chr$(0)
  CALL($1E86,UADD(ADDR(name$),1),ticks%,vol%,0,pstat%)
ENDP

PROC playc:
  CALL($2086)
ENDP

PROC playw%:(inname$,ticks%,vol%)
  LOCAL name$(128),p%,ret%
  p%=PEEKW($1c)+6             REM address of saved flags after CALL
  name$=inname$+chr$(0)
  ret%=CALL($1F86,UADD(ADDR(name$),1),ticks%,vol%)
  IF PEEKW(p%) AND 1          REM carry set for error
    RETURN ret% OR $FF00      REM return error
  ENDIF
ENDP
```

`recorda:(pstat%,inname$,size%)` and `recordw%:(inname$,size%)` respectively perform asynchronous and synchronous recording to file `inname$`. Any existing file is replaced. You can only record to the Internal disk or to a RAM SSD - you cannot record to a Flash SSD. (You can playback from a Flash SSD, however.)

`size%` specifies the maximum number of bytes to be recorded in units of 2048 bytes. To record for one second `size%=4`. This figure excludes the 32-byte header. Before recording, a file of length `32+size%*2048` bytes is created and there must actually be room on the disk for a file of that length.

pstat% is the address of the status word to take the completion code for asynchronous recording.

recordc: cancels recording and truncates the file to the actual length recorded before cancellation.

playa:(pstat%,inname$,ticks%,vol%) and playw%:(inname$,ticks%,vol%) respectively perform asynchronous and synchronous playing of inname$.

inname$ should either be the filename of the sound file to play or a '*' followed by just the name component of the sound file. If it is preceded by a '*', the extension .WVE is assumed and the service automatically searches ROM:: and the \WVE directories of M: (Internal disk), A: and B: (in that order). The ROM:: .WVE files have names SYS$AL01, SYS$AL02 and SYS$AL03.

ticks% is the duration that the sound file will play back in system ticks. If it is shorter than the given sound then playback is truncated to that time. If ticks% is negative, in addition to truncating the playback of longer files, it pads out as necessary to that duration with silence. If duration is zero, it plays the file without truncation or padding. (Alarms use a parameter of -480 to truncate or pad out to 15 seconds.)

volume% is a number between 0 and 5 inclusive, with 0 being the loudest. On the Series 3c there are only 4 actual levels: 0/1, 2, 3, and 4/5. pstat% is the address of the status word to take the completion code for asynchronous playback. Playback will append periods of silence and repeat the sound as specified in the file header.

playc: cancels playing back a sound.

Alarms

The system dialogs that are used to set alarms detect the presence of any .WVE files in the \WVE directory of any local directory (in practice, A:, B: and Internal) and make these available (by file name) as the sound of the alarm.

When an alarm with a .WVE file rings, .WVE file playback (including any repeats) is clipped to 15 seconds. If the .WVE file plays for less than 15 seconds (including any repeats), the 15 seconds is padded out with silence.

Example of recording

The following asynchronously records time% seconds of sound to file file$ or cancels the recording when any key is pressed. The section I/O device handling and asynchronous requests also in this chapter discusses the principles involved. The recorda: and recordc: procedures, from above, are used.

```
PROC record:(file$,time%)
  LOCAL sstat%,kstat%,key%(4),size%,ret%,signals%
  size%=time%*4
  recorda:(ADDR(sstat%),file$,size%)            REM async record
  IOC(-2,1,kstat%,key%())                       REM async key read
  WHILE 1
    IOWAIT              REM wait for recording to complete, or a key
    IF sstat%<>-46                REM if sound no longer pending
      IOCANCEL(-2)                REM cancel key read
      IOWAITSTAT kstat%           REM wait for cancellation
      IF sstat%<0
        gIPRINT "Error recording:"+err$(sstat%)
      ENDIF
      BREAK
```

```
    ELSEIF kstat%<>-46              REM else if key pressed
      recordc:                      REM cancel record
      IOWAITSTAT sstat%             REM wait for cancellation
      gIPRINT "Cancelled"
      BREAK
    ELSE                 REM some async request made outside this PROC
      signals%=signals%+1           REM save it for later
    ENDIF
  ENDWH
  WHILE signals%
    IOSIGNAL                        REM put back foreign signals
    signals%=signals%-1
  ENDWH
ENDP
```

Î **The following section does not apply to the Series 5.**

Series 3c and Siena OPL database information

`ODBINFO var info%()` is provided for advanced use only and allows you to use OS and CALL to call *DbfManager* interrupt functions not accessible with other OPL keywords.

The description given here will be meaningful only to those who have access to full SDK documentation of the DbfManager services, which explains any new terms. Since that documentation is essential for use of ODBINFO, no attempt is made here to explain these terms.

ODBINFO returns `info%()`, which must have four elements containing pointers to four blocks of data; the first corresponds to the file with logical name `A`, the second to `B` and so on.

Take extreme care not to corrupt these blocks of memory, as they are the actual data structures used by the OPL runtime interpreter.

A data block which has no open file using it has zero in the first two bytes. Otherwise, the block of data for each file has the following structure, giving the offset to each component from the start of the block and with offset 0 for the 1st byte of the block:

Offset	*Bytes*	*Description*
0	2	DBF system's file control block (handle) or zero if file not open
2	2	offset in the record buffer to the current record
4	2	pointer to the field name buffer
6	2	number of fields
8	2	pointer to start of record buffer
10	2	length of a NULL record
12	1	non-zero if all fields are text
13	1	non-zero for read-only file
14	1	non-zero if record has been copied down
15	1	number of text fields

16 2 pointer to device name

EXAMPLE

To copy the *Descriptive Record* of logical file B to logical file C:

```
PROC dbfDesc:
  LOCAL ax%,bx%,cx%,dx%,si%,di%
  LOCAL info%(4),len%,psrc%,pdest%
  ODBINFO info%()
  bx%=PEEKW(info%(2))              REM handle of logical file B
  ax%=$1700                        REM DbfDescRecordRead
  IF OS($d8,ADDR(ax%)) and 1
    RETURN ax% OR $ff00            REM return the error
  ENDIF
  REM the descriptive record has length ax%
  REM and is at address peekW(uadd(info%(2),8))
  IF ax%=0
    RETURN 0                       REM no DescRecord
  ENDIF
  len%=ax%+2            REM length of the descriptive record read
                        REM + 2-byte header
  psrc%=PEEKW(uadd(info%(2),8))
  pdest%=PEEKW(uadd(info%(3),8))
  CALL($a1,0,len%,0,psrc%,pdest%)  REM copy to C's buffer
  cx%=len%
  bx%=PEEKW(info%(3))              REM handle of logical file C
  ax%=$1800                        REM DbfDescRecordWrite
  IF OS($d8,ADDR(ax%)) and 1
    RETURN ax% OR $ff00
  ENDIF
  RETURN 0                         REM success
ENDP
```

Î **The following section does not apply to the Series 5.** OPXs provide a different mechanism for calling language extensions and creating object instances. It uses language extensions provided in separate EPOC32 DLLs written especially for OPL support. These DLLs can be added to the language by anyone at any time and have the file extension .OPX. Unlike procedures written in OPL, these procedures are as fast to call as built-in keywords. OPXs enable OPL programs to perform virtually any operation in EPOC32 which is available to a C++ program. OPX procedures are called in a similar way to user-defined OPL procedures.

Series 3c and Siena DYL handling

This section contains a complete reference description of OPL's support for accessing previously created dynamic libraries (*DYLs*). These libraries have an object-oriented programming (*OOP*) user-interface and several have been built into the Series 3c ROM for use by the ROM applications. DYLs cannot be created using OPL.

Since a vast amount of documentation would need to be provided to describe the essential concepts of OOP and the services available in existing DYLs, no attempt is made to supply it here. This section simply introduces the syntax for all the OOP keywords supported in OPL with a brief description of each. Also, OOP terminology is used here without explanation, to cater for those who have previous experience of DYL handling in the 'C' programming language.

var arguments

The use of `var` and `#` for arguments was discussed earlier in this chapter in the section 'I/O functions and commands'. The DYL handling keywords use `var` and `#` in the same way, for example:

```
ret%=SEND(pobj%,method%,var p1,var p2,var p3)
```

This is because many DYL methods need the address of a variable or of a structure to be passed to them.

When you use a LOCAL or GLOBAL variable as the `var` argument, the address of the variable is used. (You cannot use procedure parameters or field variables, for this reason.) **If you use a `#` before a `var` argument, though, the argument/value is used directly, instead of its address being used.**

If, for example, you need to call a method with `p1` the **address** of a long variable `a&`, `p2` the integer constant 3, and `p3` the address of a zero terminated string `"X"`, you could call it as follows:

```
s$="X"+CHR$(0)                          REM zero terminate
p%=UADD(ADDR(s$),1)                     REM skip leading count byte
ret%=SEND(pobj%,method%,a&,#3,#p%)
```

The address of `a&` is passed because there is no `#`. `3` and the value in `p%` are passed directly (no address is taken) because they are preceded by `#`.

Loading a DYL

`ret%=LOADLIB(var cathand%,name$,link%)` loads and optionally links a DYL that is not in the ROM. If successful, writes the category handle to `cathand%` and returns zero. You would normally only set `link%` to zero if the DYL uses another DYL which you have yet to load in which case LINKLIB would subsequently be used. The DYL is shared in memory if already loaded by another process.

Unloading a DYL

`ret%=UNLOADLIB(cathand%)` unloads a DYL from memory. Returns zero if successful.

Linking a DYL

`LINKLIB cathand%` links any libraries that have been loaded using LOADLIB. LINKLIB is not likely to be used much in OPL - pass `link%` with a non-zero value to LOADLIB instead.

Finding a category handle given its name

`ret%=FINDLIB(var cathand%,name$)` finds DYL category `name$` (including `.DYL` extension) in the ROM. On success returns zero and writes the category handle to `cathand%`. To get the handle of a RAM-based DYL, use LOADLIB which guarantees that the DYL remains loaded in RAM. FINDLIB will get the handle of a RAM-based DYL but does not keep it in RAM.

Converting a category number to a handle

`cathand%=GETLIBH(catnum%)` converts a category number `catnum%` to a handle. If `catnum%` is zero, this gets the handle for OPL.DYL.

Creating an object by category number

`pobj%=NEWOBJ(catnum%,clnum%)` creates a new object by category number `catnum%` belonging to the class `clnum%`, returning the object handle on success or zero if out of memory. This keyword simply converts the category number supplied to a category handle using GETLIBH and then calls NEWOBJH.

Creating an object by category handle

`pobj%=NEWOBJH(cathand%,clnum%)` creates a new object by category handle `cathand%` belonging to the class `clnum%`, returning the object handle on success or zero if out of memory.

Sending a message to an object

```
ret%=SEND(pobj%,method%)
ret%=SEND(pobj%,method%,var p1)
ret%=SEND(pobj%,method%,var p1,var p2)”
ret%=SEND(pobj%,method%,var p1,var p2,var p3)
```

send a message to the object `pobj%` to call the method number `method%`, passing between zero and three arguments depending on the requirements of the method, and returning the value returned by the selected method.

Protected message sending

```
ret%=ENTERSEND(pobj%,method%)
ret%=ENTERSEND(pobj%,method%,var p1)
ret%=ENTERSEND(pobj%,method%,var p1,var p2)
ret%=ENTERSEND(pobj%,method%,var p1,var p2,var p3)
```

send a message to an object with protection.

Methods which return errors by *leaving* must be called with protection.

ENTERSEND is the same as SEND except that, if the method leaves, the error code is returned to the caller; otherwise the value returned is as returned by the method.

Use ENTERSEND0 (described next) for methods which leave but do not return a value explicitly on success.

Protected message sending (returns zero on success)

```
ret%=ENTERSEND0(pobj%,method%)
ret%=ENTERSEND0(pobj%,method%,var p1)
ret%=ENTERSEND0(pobj%,method%,var p1,var p2)
ret%=ENTERSEND0(pobj%,method%,var p1,var p2,var p3)
```

send a message to an object with protection and guarantee that the known value zero is returned on success. Otherwise ENTERSEND0 is the same as ENTERSEND.

Methods which return errors by *leaving* but return nothing (or *NULL*) on success must use ENTERSEND0. Besides providing protection, ENTERSEND0 also returns zero if the method did not leave, or the negative error code if it did.

If ENTERSEND were incorrectly used instead and the method completed successfully (i.e. without leaving), the return value would be random and could therefore be in the range of the error codes implying that the method failed.

Dynamic memory allocation

For each running OPL program (or *process*) the operating system automatically allocates memory. On the Series 3c, this can grow up to a maximum of 32 bytes less than 64K. On the Series 5, there are no built-in memory limits and a module or application can use as much of the available memory as it requires. A corollary of this is that although the use of integers were sufficient for the storage of addresses on the Series 3c, this is no longer the case on the Series 5. Hence long integers must be used instead. See the '32-bit addressing' section below.

Maximum data size in a procedure

Although there is no built-in limit to the total amount of data in an OPL program on the Series 5, it is still not possible to declare variables in a procedure that use in total more than 65516 bytes. This allows the largest integer array in a procedure to have 32757 elements. This number decreases for any other variables, externals or for procedures called from the given procedure, as they all consume some of the procedure's data space.

The maximum for the Series 3c is about 32758 bytes of data in total per procedure.

Series 5 32-bit addressing

As described above, memory addresses outside the 64K address space need to be stored in long integers. Thus all keywords which support addresses take and return long integers on the Series 5. Note also that where `#` is used to specify the address of a variable that is listed in the syntax using `var`, the address is a long integer, so you would use `#address&`. See the 'I/O functions and commands' section above.

Porting Series 3c programs to the Series 5

To facilitate porting of OPL programs written on the Series 3c (and other earlier machines) to the Series 5, it is in fact possible to set a flag to emulate the Series 3c memory model using `SETFLAGS 1`. This will cause an 'Out of memory' error if an attempt is made to obtain an address beyond the 64K limit.

If this flag is set, the Series 5 checks that the 64K limit is not exceeded when,

- the variables for a procedure are allocated, on calling any procedure. If the address of any variable in the procedure would require more than 16 bits, the procedure will fail to be loaded and an 'Out of memory' error will be raised. This is evidently preferable to having, for example, `p%=ADDR(i%)` giving an 'Overflow' error, once the procedure has been loaded. The ideal solution is, of course, to port the program to use 32-bit addresses instead, but setting the flag is a quick solution for many applications that are known to require less than 64K.

- the value returned by the heap allocation keywords requires more than 16 bits.

Some existing Series 3c programs may at times attempt to exceed the 64K limit and deal with the 'Out of memory' error. To port such programs there are two choices. Either you can

- change all integers that contain addresses to be 32-bit

or

- set the flag that enforces the 64K limit using SETFLAGS.

Note that the use of the word "address" is in fact imprecise for the Series 5. Series 5 addresses are in fact offsets into OPL's variable-heap, so an address of 0 really means 0 offset relative to the base of OPL's heap. This allows OPL to perform the appropriate 64K limit tests. This method of addressing variables is referred to as "base-relative addressing".

Another potential problem may well concern you at this point. When, for example, displaying an OPL address in some diagnostic debugging code, if a variable `i%` or a heap cell has a base-relative offset of

between 32768 and 65535, then `p%=ADDR(i%)` would produce an overflow error unless special measures were taken. The reason for this is that 16-bit integers are *signed* in OPL, so the range is -32768 to 32767. Hence, although 32768 fits in an *unsigned* 16-bit integer (hex $8000) it doesn't fit into a *signed* integer. OPL gets around this problem by *sign-extending* the result of ADDR and the heap functions when 64K restriction is in force, as follows:

- if the value returned is 32768 to 65535 (hex $8000 to $ffff), OPL treats the value as signed. So $8000 becomes &ffff8000 (or -32768), and $ffff becomes &ffffffff (or -1). These values can then be assigned to the signed `p%`.

- if the value is greater than or equal to 65536 (hex &10000) then it is not sign-extended and so assigning it to `p%` raises an 'Overflow' error as required.

As mentioned above, this conversion should always be totally transparent to your program. This is possible because the keywords that use these sign-extended addresses, all convert the value back to an unsigned 16-bit value before use.)

There is one case, however, where porting is required even with the flag set for a 64K limit. This is when the long integer value returned by ADDR or from a heap-allocating function is passed directly into a user-defined procedure. An OPL procedure from the Series 3c taking an address parameter will probably have been written to take a 16-bit address. As the translator cannot necessarily know the type taken by a user-defined procedure, it cannot coerce the type returned by these functions to match that taken by the user-defined procedure. This means that a type violation error can be caused.

Example:

If a user-defined procedure, `userProc:(p%)` is called as follows:

```
userProc:(ADDR(i%))
```

the 32-bit integer returned by ADDR will cause a type violation, as it will not be coerced to a 16-bit integer. The solution is either to define `userProc:` to take a 32-bit integer, or to declare the prototype of `userProc:` as `userProc:(ADDR%)`. Either of these would allow translator coercion. See the 'Calling Procedures' chapter and also EXTERNAL in the 'Alphabetic Lisiting' for more information on procedure prototyping in OPL on the Series 5.

Porting Series 3c programs to the Series 5 without enforcing the 64K limit

If you are porting Series 3c OPL code (or code from earlier machines) to the Series 5 without using a SETFLAGS statement, it should be obvious that there are two points which you need to remember:

- any address variables which were previously specified as integers must be changed to long integers.

- as mentioned in the 'Safe pointer arithmetic' section above, UADD and USUB should **not** be used when you are using 32-bit addressing without the flag set to restrict to 64K. Use of these functions should be replaced with ordinary long integer arithmetic.

Overview of memory usage

The actual memory used by an OPL program depends on the requirements of the process and is automatically grown or shrunk as necessary.

Î An OPL process contains several separate memory areas, but the only one of significant interest to the OPL programmer is the *OPL heap*, used to contain the OPL variables and dynamically allocated memory cells. The heaps of different processes are entirely separate - you need only concern yourself with the heap used in your own process.

Ì This memory is called the *process data segment* and contains all the data used by the process as well as some fixed length data at low memory in the segment needed by the operating system to manage the process and for other system data.

Although the data segment for an OPL process contains several components, the only component of significant interest to the OPL programmer is the *process heap*. This section describes several keywords for accessing the heap.

The heap is essentially a block of memory at the highest addresses in a process data segment, so that the operating system can grow and shrink the heap simply by growing and shrinking the data segment and without having to move other blocks of memory at higher addresses in the data segment. The heaps of different processes are totally independent - you need concern yourself only with the heap used in your own data segment.

The heap allocator

The heap allocator keywords are used to allocate, resize and free variable length memory cells from the OPL heap (the process heap on the Series 3c). Cells typically range in size from tens of bytes to a few kilobytes. Allocated cells are referenced directly by their address; they do not move to compact free space left by freed cells.

Heap allocator keywords are:

- ALLOC allocates a cell of specified size, returning its address.
- FREEALLOC frees a previously allocated cell, which is returned to the heap.
- REALLOC changes the size of a cell, returning its new address.
- ADJUSTALLOC opens or closes a gap in the middle of a cell (useful for insertion or deletion of cell content), changing the size of the cell as appropriate.
- LENALLOC returns the size of a cell.

The heap structure

Initially, the heap consists of a single free cell. After a number of calls to allocate and free cells, the heap typically consists of ranges of adjacent allocated cells separated by single free cells (which are linked). If a cell being freed is next to another free cell the two cells are automatically joined to make a single cell to prevent the free cell linked list from growing unnecessarily.

Writing beyond the end of a cell will corrupt the heap's integrity. Such errors are difficult to debug because there is no immediate effect - the corruption is a "time bomb". It will eventually be detected, resulting in the process exiting prematurely by a subsequent allocator call such as FREEALLOC.

Growing and shrinking the heap

The heap is not fixed in size. The operating system can grow the heap to satisfy allocation requests or shrink it to release memory back to the system.

Allocation of cells is based on "walking" the free space list to find the first free cell that is big enough to satisfy the request. If no free cell is big enough, the operating system will attempt to grow the data segment to add more free space at the end of the heap.

If there is no memory in the system to accommodate growth or, on the Series 3c and Siena, if the data segment has reached its maximum size of (approximately) 64K, the allocate request fails. There are few circumstances when an allocate request can be assumed to succeed and calls to ALLOC, REALLOC and ADJUSTALLOC should have error recovery code to handle a failure to allocate.

Lost cells

There are cases in which programs allocate a sequence of cells which must either exist as a whole or not at all. If during the allocate sequence one of the later allocations fails, the previously allocated cells must be freed. If this is not done, the heap will contain unreferenced cells that consume memory to no purpose.

When designing multi-cell sequences of this kind, you should be mindful of the recovery code that must be written to free partially built multi-cell structures. The fewer the cells in such a structure, the simpler the recovery code is.

Internal fragmentation

The free space in the heap is normally fragmented to some extent; the largest cell that can be allocated is substantially smaller than the total free space. Excessive fragmentation, where the free space is distributed over a large number of cells - and where, by implication, many of the free cells are small - should be avoided because it results in inefficient use of memory and reduces the speed with which cells are allocated and freed.

Practical design hints for limiting internal fragmentation are:

- Avoid using the heap for small, highly transient data structures for which ordinary variables are adequate. High frequency cycling through allocate and free pairs, “churns” the heap and leads to a long free space list.
- When you have a large number of variable length data structures - particularly when they are frequently resized, “granularise” them (i.e. round the allocation up to a multiple of some reasonable value) so that you decrease the chance of leaving small, unusable free space cells.

Although a small number of gaps are not too serious and should eventually disappear in most cases anyway, the new heap allocating keywords provide ample opportunity to fragment the heap. Provided that you create and free cells in a careful and structured way, where any task needing the allocator frees them tidily on completion, there should not be a problem.

The OPL runtime interpreter and the heap

Ì The OPL runtime interpreter, which actually runs your program, uses the same data segment and heap as your program and makes extensive use of the heap. It is very important that you should understand the interpreter’s use of the heap - at least to a limited extent to avoid substantial internal fragmentation as described above.

Whenever an OPL procedure is called, a cell is allocated to store data required by the interpreter to manage the procedure. The same cell contains all the variables that you have declared in the procedure. On the Series 3c, when cacheing is not being used, the same cell also contains the translated code for the procedure which is interpreted. When the procedure returns (or implicitly returns due to an error) the cell is freed again back to the heap. This use of the heap is very tidy - adjacent cells are allocated and freed with little opportunity for leaving gaps in the heap.

Unfortunately, various other keywords also cause cells to be allocated and these can cause fragmentation. For example, LOADM, CREATE, OPEN etc. all allocate cells; UNLOADM, CLOSE etc. free those cells. If a procedure is called which uses CREATE to create a data file, the procedure cell is allocated, followed by the CREATE cell and the procedure cell is then freed when the procedure returns. The heap structure therefore contains a gap where the procedure cell was, which remains until all cells at higher addresses are freed.

Î On the Series 5, the OPL runtime interpreter uses two separate heaps: one for process management and one for user variables and allocated cells. LOADM, CREATE, OPEN, etc use the process management heap. This ensures that the use of these keywords will not cause fragmentation. It is therefore not necessary for you to understand the interpreter’s use of the heap.

Warning - peeking/poking the cell

Using the allocator is by no means simple in OPL since the data in an allocated cell usually has to be read or written in OPL using the PEEK and POKE set of keywords which are intrinsically subject to programming error. OPL does not provide good support for handling pointers (variables containing addresses), which are basic to heap usage, nor for complicated data structures, so that it is all too easy to make simple programming errors that have disastrous effects.

For these reasons, you are recommended to use the heap accessing keywords only when strictly necessary (which should not be very often) and to take extreme care when you do use them. On the other hand, for programmers with previous experience of dynamic memory allocation, the heap allocation keywords will often prove most useful.

Reasons for using the heap allocator

A few common instances where the allocator might be used are:

- when the amount of data to be stored is variable or cannot be determined at the time of writing the program. Without using the allocator, you would have to declare a large array to hold the data always even when it turns out that only a few bytes are needed in a particular case. Using the allocator allows you to grow the cell containing your data as and when required.
- the amount of data may be specified in a file or by the user of the program. Once again, you would need to declare a possibly unnecessarily large array to cope with all allowed cases.
- a system of library procedures might use a common cell, usually called a *control block*, to store common data. You could have one procedure creating the cell and initialising data in it, other procedures in the system could be passed the address of the cell, using and possibly updating the data in it, and finally a further procedure could free the cell.

  This concept will be familiar to you if you have used handles for the I/O keywords, where the handle references a cell used internally by the I/O system.

  If you did not use the allocator in this case, you would probably need to declare a global array in the procedure calling the library procedures, with the disadvantages that the name and size of the array would need to be fixed for all time even when a better alternative mechanism has been devised for the library code with different data requirements.

- ADJUSTALLOC allows you to insert or remove data at the start or in the middle of data that has previously been set up. With an array, you would need to copy each element to the next or previous element to make or close a gap.

Î ALLOC and associated heap keywords

On the Series 5, ALLOC, REALLOC and ADJUSTALLOC allocate cells that have lengths that are the smallest multiple of four greater than the size requested. All of these raise errors if the cell address argument is not in the range known by the heap. The same address checking is done for peeking and poking, but note that OPL on the Series 5 allows the addresses of the application-specific SIBO magic statics DatApp1 to DatApp7 (hex 28 to 34 inclusive) to be used for compatibility.

ALLOC, REALLOC and ADJUSTALLOC return a long integer value, so that a request can be made to allocate a cell of any length within memory constraints.

See also sections above for discussion of 32-bit addressing.

Ì Note that in the sections which follow, Series 5 32-bit addressing is assumed. If you are using a Series 3c, you can substitute integers for long integers, so for example the usage of ALLOC becomes, `pcell%=ALLOC(size%)`, i.e. `%` rather than `&`. However, using long integers will make any porting to a Series 5 machine in the future easier.

Allocating a cell

Use `pcell&=ALLOC(size&)` to allocate a cell on the heap of a specified size returning the pointer to the cell or zero if there is not enough memory. **The new cell is uninitialised** - you cannot assume that it is zeroed.

Freeing an allocated cell

Use `FREEALLOC pcell&` to free a previously allocated cell at `pcell&` as returned, for example, by ALLOC. Does nothing if `pcell&` is zero.

Changing a cell's size

Use `pcelln&=REALLOC(pcell&,size&)` to change the size of a previously allocated cell at address `pcell&` to `size&`, returning the new cell address or zero if there is not enough memory. If out of memory, the old cell at `pcell&` is left as it was.

If successful, `pcelln&` will not be the same as `pcell&` on return only if the size increases and there is no free cell following the cell being grown which is large enough to accommodate the extra amount.

Inserting or deleting data in cell

Use `pcelln&=ADJUSTALLOC(pcell&,offset&,amount&)` to open or close a gap at `offset&` within the allocated cell `pcell&` returning the new cell address or zero if there is not enough memory. `offset&` is 0 for the first byte in the cell. Opens a gap if `amount&` is positive and closes it if negative. The data in the cell is automatically copied to the new position.

If successful, `pcelln&` will not be the same as `pcell&` on return only if `amount&` is positive and there is no free cell following the cell being adjusted which is large enough to accommodate the extra amount.

Finding out the cell length

Use `len&=LENALLOC(pcell&)` to get the length of the previously allocated cell at `pcell&`.

EXAMPLE USING THE ALLOCATOR

This example illustrates the careful error checking which is essential when using the allocator. RAISE is used to jump to the error recovery code.

If you cannot understand this example it would be wise to avoid using the allocator altogether.

Ì As above, note that Series 5 32-bit addressing is assumed. If you are using a Series 3c, you could substitute integers for long integers, so for example the usage of ALLOC becomes, `pcell%=ALLOC(size%)`, i.e. `%` rather than `&`.

```
LOCAL pcell&                          REM pointer to cell
LOCAL pcelln&                         REM new pointer to cell
LOCAL p&                              REM general pointer
LOCAL n%                              REM general integer
ONERR e1
pcell&=ALLOC(2+2*8)                   REM holds an integer and 2
                                      REM 8-byte floats initially
IF pcell&=0
  RAISE -10                           REM out of memory; go to e1::
ENDIF
POKEW pcell&,2                  REM store integer 2 at start of cell
                                REM i.e. no. of floats
POKEF UADD(pcell&,2),2.72             REM store float 2.72
POKEF UADD(pcell&,10),3.14            REM store float 3.14 ...
pcelln&=REALLOC(pcell&,2+3*8) REM space for 3rd float
IF pcelln&=0
  RAISE -10                           REM out of memory
ENDIF
```

```
pcell&=pcelln&                        REM use new cell address
n%=PEEKW(pcell&)                      REM no. of floats in cell
POKEF UADD(pcell&,2+n%*8),1.0 REM 1.0 after 3.14
POKEW pcell&,n%+1             REM one more float in cell ...
pcelln&=ADJUSTALLOC(pcell&,2,8)     REM open gap before 2.72
IF pcelln&=0
  RAISE -10                           REM out of memory
ENDIF
pcell&=pcelln&                        REM use new cell address
POKEF UADD(pcell&,2),1.0              REM store 1.0 before 2.72
POKEW pcell&,4                        REM 4 floats in cell now ...
p&=UADD(pcell&,LENALLOC(pcell&))    REM byte after cell end
p&=USUB(p&,8)                         REM address of final float
POKEF p&,90000.1                      REM overwrite with 90000.1
RAISE 0                               REM clear ERR value
e1::
FREEALLOC pcell&                      REM free any cell created
IF err<>0
  ...                                 REM display error message etc.
ENDIF
RETURN ERR
```